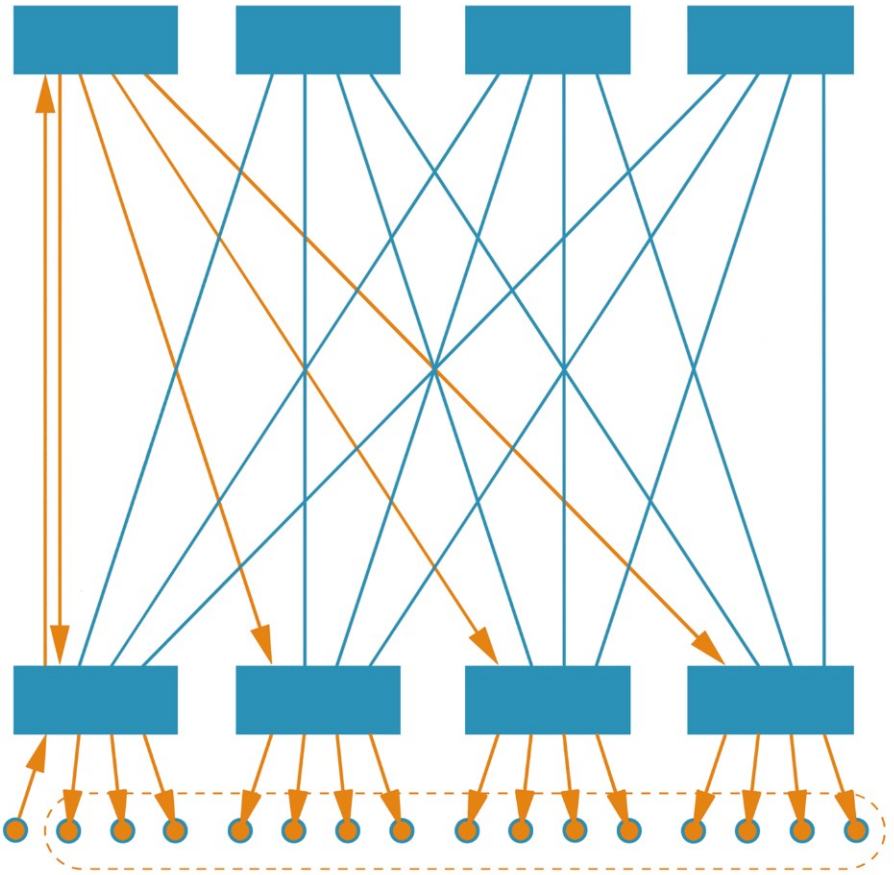
PERFORMANCE ANALYSIS AND GRID COMPUTING



Edited by
Vladimir Getov
Michael Gerndt
Adolfy Hoisie
Allen Malony
Barton Miller

PERFORMANCE ANALYSIS AND GRID COMPUTING

PERFORMANCE ANALYSIS AND GRID COMPUTING

Selected Articles from the Workshop on Performance Analysis and Distributed Computing August 19–23, 2002, Dagstuhl, Germany

Edited by

Vladimir Getov
University of Westminster, UK

Michael Gerndt
Technical University Munich, Germany

Adolfy Hoisie
Los Alamos National Laboratory, USA

Allen Malony
University of Oregon–Eugene, USA

Barton Miller
University of Wisconsin–Madison, USA

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available from the Library of Congress.

Performance Analysis and Grid Computing
Edited by Vladimir Getov, Michael Gerndt, Adolfy Hoisie, Allen Malony and Barton Miller
ISBN 978-1-4613-5038-5 ISBN 978-1-4615-0361-3 (eBook)
DOI 10.1007/978-1-4615-0361-3

Originally published by Kluwer Academic Publishers in 2004
Softcover reprint of the hardcover 1st edition 2004

Printed on acid-free paper.

Contents

Preface

Since the invention of the electronic computers, achieving higher performance of both systems and applications has always been a main concern and motivation for the research community. Given the tremendous pace of evolution in information technologies, this field demands continuous efforts for innovations in performance theory, methodologies, and tools. The joint development of both performance science and performance technology spans a rich spectrum of related research areas, such as performance modeling, evaluation, instrumentation, measurement, analysis, monitoring, interpretation, optimization, and prediction.

Past and current research in computer performance analaysis has focused primarily on dedicated parallel machines. However, future applications in the area of high-performance computing will not only use individual parallel systems but a large set of networked resources. This scenario of computational and data Grids is attracting a great deal of attention from both computer and computational scientists. In addition to the inherent complexity of parallel machines, the sharing and transparency of the available resources introduce new challenges on performance analysis, techniques, and systems. In order to meet those challenges, a multi-disciplinary approach to the multi-faceted problems of performance is required. New degrees of freedom will come into play with a direct impact on the performance of Grid computing, including wide-area network performance, quality-of-service (QoS), heterogeneity, and middleware systems, to mention only a few.

Reflecting this new reality, the Dagstuhl Workshop on Performance Analysis and Distributed Computing[1], organized by members of the European Working Group APART[2], was designed to bring together researchers from both the high-performance computing and the distributed computing areas. Indeed, attendees with expertise in a variety of topics in computer and computational sciences arrived in Dagstuhl at the end of August 2002 to discuss the state-of-the-art and

[1] http://www.dagstuhl.de/02341/
[2] http:www.fz-juelich.de/apart

the future of computer performance research as it applies to systems, applications, and Grids. The topics covered by the workshop were grouped into five areas:

1 Performance Modeling and Analysis. The contributions to this area explored performance prediction and its application in different contexts, emphasizing on distributes shared memory multiprocessors, large scale systems, task parallel programs, and Grid computing.

2 Novel Architectures. This area covered quite diverse aspects, including mobile agents, cellular architectures in the context of future high-end computer systems such as IBM's Blue Gene project, and high-performance interconnects such as the Quadrics network.

3 Performance Tools and Systems. Two key issues were addressed in this area, namely scalability of performance analysis tools, and tool automation. Other presentations covered performance analysis for Java, performance visualization, and data management.

4 Grid Performance. The presentations in this area gave an overview of the current approaches to performance monitoring and analysis in several Grid-related European projects, such as CrossGrid, UNICORE, DataGrid, and DAMIEN.

5 Grid Computing and Applications. The presentations concentrated on programming aspects of Grid computing infrastructures and Grid applications. In addition, performance aspects of Web and Grid services were introduced and analyzed using application case studies.

The presentations during the seminar led to two major threads of evening gatherings and discussions centered on *Grid Computing* and *Future Architectures*. The main open questions addressed in the Grid Computing discussion were:

- Will QoS become a reality in the context of Grids? Most of the participants expressed the opinion that this will depend significantly on economic issues. If users pay for using the Grid services, performance guarantees would be required.

- What are the new aspects of performance optimization in this area when applied to computational Grids? In Grid environments, performance optimization focuses mainly on efficient scheduling decisions in dynamic and heterogeneous infrastructures, dynamic performance tuning, and performance steering.

- Does the analogy between the Grid computing and the electrical power grid hold and, if yes, how different are they? Two main differences were identified: first, users transfer information via the Grid which raises major security problems, and second, the variety of Grid services and resources is much richer, which makes performance steering, resource management, and accounting very important topics for research and development.
- How will Grids be used? The majority of attendees favored the concept of virtual organizations as the main usage of the Grid as opposed to the metacomputing style of applications. The main programming paradigm for Grid computing could follow a components-based approach.

In the area of future architectures, the discussions centered around fundamental and practical topics likely to impact the development of next generation systems, such as:

- How will the available chip space be used? Can scalar and vector processors be combined?
- What will be the impact of recent research on processor in memory systems, multiprocessor, and multi-threading architectures on the future high-end computers?
- When will Quantum Computing become a reality? The prevailing opinion seemed to be that the development of this technology might take approximately another 50 years.
- What will be the role of reconfigurable architectures in the future distributed and Grid computing systems?

The selected contributions which we are proud to preface here, are an encapsulation of a scientifically rich event. The articles are organized in three parts: Performance Modeling and Analysis, Performance Tools and Systems, and Grid Performance and Applications.

To conclude, we would like to thank all the participants for their contribution to making the workshop a resounding success; the staff of the International Conference and Research Center for Computer Science at Schloss Dagstuhl for their professional support in the organization; and, last but not least, all the authors that contributed papers for publication in this volume.

Our thanks also go to the European Commission for sponsoring this volume of selected articles from the workshop via the European Working Group APART (Automatic Performance Analysis: Real Tools), project number IST-2000-28077.

VLADIMIR GETOV, MICHAEL GERNDT, ADOLFY HOISIE, ALLEN MALONY, BARTON MILLER

Contributing Authors

Cliff Addison Computing Services Department, University of Liverpool, Liverpool L69 3BX, UK (caddison@liv.ac.uk)

Bartosz Baliś Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland (balis@uci.agh.edu.pl)

Robert Bell Computer and Information Science Department, University of Oregon, Eugene, OR 97403, USA (bertie@cs.uoregon.edu)

Sarah Bennett Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK (s.bennett@imperial.ac.uk)

Marian Bubak Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland (bubak@uci.agh.edu.pl)

Nikos Chrisochoides Department of Computer Science, College of William and Mary, P.O. Box 8795, Williamsburg, VA 23187, USA (nikos@cs.wm.edu)

Salvador Coll Department of Electronic Engineering, Technical University of Valencia, Camino de Vera, 46022 Valencia, Spain (scoll@eln.upv.es)

Catherine Crawford IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA (catcraw@us.ibm.com)

Daniel Dias IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA (dias@us.ibm.com)

José Duato Department of Electronic Engineering, Technical University of Valencia, Camino de Vera, 46022 Valencia, Spain (jduato@eln.upv.es)

Thomas Fahringer Institute for Software Science, University of Vienna, Liechtenstein Str. 22, A-1090 Vienna, Austria (tf@par.univie.ac.at)

Rupert Ford Centre for Novel Computing, University of Manchester, Manchester M13 9PL, UK (rupert@cs.man.ac.uk)

Len Freeman Centre for Novel Computing, University of Manchester, Manchester M13 9PL, UK (lfreeman@cs.man.ac.uk)

Włodzimierz Funika Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland (funika@uci.agh.edu.pl)

Vladimir Getov School of Computer Science, University of Westminster, Watford Rd, Harrow, London HA1 3TP, UK (V.S.Getov@wmin.ac.uk)

Sven Haubold Center for High Performance Computing (ZHR), Technical University of Dresden, 01062 Dresden, Germany (haubold@zhr.tu-dresden.de)

Adolfy Hoisie Performance and Architectures Laboratory, CCS-3, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545, USA (hoisie@lanl.gov)

Arun Iyengar IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA (aruni@us.ibm.com)

Josep Jorba Computer Science Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain (josep.jorba@uab.es)

Paul H J Kelly Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK (p.kelly@imperial.ac.uk)

Darren J. Kerbyson Performance and Architectures Laboratory, CCS-3, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545, USA (djk@lanl.gov)

Matthias Korch Department of Mathematics and Physics, University of Bayreuth, 95440 Bayreuth, Germany (matthias.korch@uni-bayreuth.de)

Matthias Kühnemann Department of Computer Science, Technical University of Chemnitz, 09107 Chemnitz, Germany (kumat@informatik.tu-chemnitz.de)

Craig Lee The Aerospace Corporation, 2350 E. El Segundo Blvd., El Segundo, CA 90245, USA (lee@aero.org)

Kukjin Lee Electrical and Computer Engineering Department, Iowa State University, IA 50010, USA (leekukji@iastate.edu)

Kai Li Computer and Information Science Department, University of Oregon, Eugene, OR 97403, USA (likai@cs.uoregon.edu)

Li Li Computer and Information Science Department, University of Oregon, Eugene, OR 97403, USA (lili@cs.uoregon.edu)

Bruce Lowekamp Department of Computer Science, College of William and Mary, P.O. Box 8795, Williamsburg, VA 23187, USA (lowekamp@cs.wm.edu)

Mikel Luján Centre for Novel Computing, University of Manchester, Manchester M13 9PL, UK (mlujan@cs.man.ac.uk)

Emilio Luque Computer Science Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain (emilio.luque@uab.es)

Allen D. Malony Computer and Information Science Department, University of Oregon, Eugene, OR 97403, USA (malony@cs.uoregon.edu)

Tomàs Margalef Computer Science Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain (tomas.margalef@uab.es)

Beniamino Di Martino Department of Information Engineering, Seconda Università di Napoli, Via Roma, 29 - 81031 Aversa (CE), Italy (beniamino.dimartino@unina.it)

Ken Mayes Centre for Novel Computing, University of Manchester, Manchester M13 9PL, UK (ken@cs.man.ac.uk)

Hartmut Mix Center for High Performance Computing (ZHR), Technical University of Dresden, 01062 Dresden, Germany (mix@zhr.tu-dresden.de)

Francisco J. Mora Department of Electronic Engineering, Technical University of Valencia, Camino de Vera, 46022 Valencia, Spain (fjmora@eln.upv.es)

Anna Morajko Computer Science Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain (ania@aows10.uab.es)

Oleg Morajko Computer Science Department, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain (olegm@aia.ptv.es)

Wolfgang E. Nagel Center for High Performance Computing (ZHR), Technical University of Dresden, 01062 Dresden, Germany (nagel@zhr.tu-dresden.de)

Marcos Novaes IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA (mnovaes@us.ibm.com)

Shawn D. Pautz Transport Methods, CCS-4, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545, USA (pautz@lanl.gov)

Fabrizio Petrini Performance and Architectures Laboratory, CCS-3, Los Alamos National Laboratory, P.O. Box 1663, Los Alamos, NM 87545, USA (fabrizio@lanl.gov)

Achal Prabhakar Department of Computer Science, University of Houston, TX 77204, USA (achal@cs.uh.edu)

Omer F. Rana Department of Computer Science, Cardiff University, PO Box 916, Cardiff CF24 3XF, UK (o.f.rana@cs.cf.ac.uk)

Thomas Rauber Department of Mathematics and Physics, University of Bayreuth, 95440 Bayreuth, Germany (rauber@uni-bayreuth.de)

Graham D. Riley Centre for Novel Computing, University of Manchester, Manchester M13 9PL, UK (griley@cs.man.ac.uk)

Mathilde Romberg Central Institute for Applied Mathematics (ZAM), Research Center Juelich, 52425 Juelich, Germany (m.romberg@fz-juelich.de)

Diane T. Rover Electrical and Computer Engineering Department, Iowa State University, IA 50010, USA (drover@iastate.edu)

Gudula Rünger Department of Computer Science, Technical University of Chemnitz, 09107 Chemnitz, Germany (ruenger@informatik.tu-chemnitz.de)

Clovis Seragiotto, Jr. Institute for Software Science, University of Vienna, Liechtenstein Str. 22, A-1090 Vienna, Austria (clovis@par.univie.ac.at)

Sameer Shende Computer and Information Science Department, University of Oregon, Eugene, OR 97403, USA (sameer@cs.uoregon.edu)

Tomasz Szepieniec Academic Computer Center – CYFRONET-AGH, Nawojki 11, 30-950 Kraków, Poland (t.szepieniec@cyfronet.krakow.pl)

Nick Trebon Computer and Information Science Department, University of Oregon, Eugene, OR 97403, USA (ntrebon@cs.uoregon.edu)

Roland Wismüller LRR-TUM – Technische Universität München, 80290 München, Germany (wismuell@in.tum.de)

Kwok Yeung Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, UK (kcy@doc.imperial.ac.uk)

Li Zhang IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA (zhangli@us.ibm.com)

I

PERFORMANCE MODELING AND ANALYSIS

DIFFERENT APPROACHES TO AUTOMATIC PERFORMANCE ANALYSIS OF DISTRIBUTED APPLICATIONS

Tomàs Margalef, Josep Jorba, Oleg Morajko, Anna Morajko, Emilio Luque
Computer Science Department
Universitat Autònoma de Barcelona
08193 Bellaterra, Spain
{tomas.margalef,josep.jorba,olegm,ania,emilio.luque}@uab.es

Abstract Parallel computing is a promising approach that provides more powerful computing capabilities for many scientific research fields to solve new problems. However, to take advantage of such capabilities it is necessary to ensure that the applications are successfully designed and that their performance is satisfactory. This implies that the task of the application designer does not finish when the application is free of functional bugs, and that it is necessary to carry out some performance analysis and application tuning to reach the expected performance. This application tuning requires a performance analysis, including the detection of performance bottlenecks, the identification of their causes and the modification of the application to improve behavior. These tasks require a high degree of expertise and are usually time consuming. Therefore, tools that automate some of these tasks are useful, especially for non-expert users. In this paper, we present three tools that cover different approaches to automatic performance analysis and tuning. In the first approach, we apply static automatic performance analysis. The second is based on run-time automatic analysis. The last approach sets out dynamic automatic performance tuning.

Keywords: automatic performance analysis, dynamic tuning, distributed computing

1. Introduction

In recent years, research in many fields of science has evolved new methods that require high computing capabilities. Physicists, biologists, chemists, etc. develop and submit applications with high computing demands to parallel, distributed or even Grid systems with the objective of obtaining the results in the shortest possible time and of considering the largest possible problem size. Therefore, performance is a key aspect in parallel/distributed computing. To obtain an efficient application, it is necessary to consider two main issues:

- The design and development of the application.

- Application performance analysis and tuning.

Design and development of distributed applications require a detailed knowledge of the system's features to take advantage of its capabilities. Moreover, distributed systems usually involve heterogeneous systems, and this fact further complicates the design of efficient applications. However, in many cases the developers of these high performance systems are not computer specialists and, typically, they are not interested in the low-level details of their systems. Therefore, many libraries and software layers have been developed to facilitate the construction of distributed applications. The main concern is that, in many cases, the software layers that have been introduced imply degradation in the performance of the application, due to the general conception of such software.

Once an application has been developed, and has been tested from the functional point of view, the programmers must investigate its performance. Therefore, they carry out some performance analysis to assess the behavior of the application, to detect potential performance bottlenecks, and to determine their causes. Finally, the programmers modifies the application, according to their best knowledge to overcome the problems identified.

It must be pointed out that, in practice, the causes of performance bottlenecks can be found at different levels. For example, a communication problem can result from:

- An erroneous conception of the application that provokes an unnecessary blocking time in a receive primitive.
- Communication library implementation. In many cases, the design or implementation of the software layers is generic and is not optimized for a particular system or for particular conditions. This implies that the application may behave differently than expected.
- Operating system features. For example, an inappropriate buffer size and the message treatment at the protocol level can interfere with application message delivery times.
- Underlying hardware capabilities. The interconnection network features (latency, bandwidth, etc.) or even the contention in the network can seriously slow down the application.

Moreover, in many cases the application performance depends on the input data set. This fact implies that a set of potential bottlenecks can vary for different executions.

As a consequence, performance analysis is a difficult and costly task. The developers are forced to master the application, the involved software layers and the distributed system behavior, and this can be too complex for non-specialists.

To tackle all these problems, user-friendly tools should be available. The required tools include programming environments, debugging tools and performance analysis systems. However, in the area of performance analysis,

there is still a lack of real useful tools and most of what is available require a high degree of expertise from the user.

The classical approach for performance analysis and tuning was based on visualization tools. First, programmers develop and debug their applications. Next, they run them with the help of a monitoring tool that collects the information on the behavior of the application. Then, in a post-mortem phase, a visualization tool shows the collected information using different views (such as gantt charts, bar charts, pie charts, etc.). There are several visualization tools available [1–3] that offer fairly evolved interfaces and allow the user to navigate amongst the different views, providing quite intuitive screens. With these tools, it is reasonably easy to gain quick insight into the behavior of the application and to find main performance bottlenecks.

However, a more difficult task is that of identifying the real causes of the bottlenecks, and determining what should be modified in the application source code, or in the system itself, to overcome them. In the classical approach, these issues are not addressed. Therefore, in recent years the interest in automatic performance analysis has significantly increased. In the automatic approach, the tools guide the programmer in the performance improvement phase by searching for the causes of bottlenecks and by providing useful recommendations on how to solve them. Several tools for approaching automatic performance analysis were developed or are currently under development [4–7].

In this work, we present three different approaches to automatic performance analysis. Each approach has target applications, with specific features, and is intended to satisfy specific user needs, according to user capabilities.

The remainder of this paper is organized as follows. Section 2 introduces and discusses three studied approaches to automatic performance analysis. The first approach – a static automatic performance analysis – is described in Section 3. In this approach, the analysis is performed in a post-mortem phase and can consider all detailed information gathered during application execution. This approach is suitable for applications that do not present variable or dynamic behavior depending on the input data set. In Section 4, we present a dynamic automatic performance analysis that avoids trace files and can control the amount of instrumentation inserted in the application. This approach is suitable for large applications with a stable behavior. Section 5 describes the third approach, a dynamic automatic performance tuning, which tries to adapt the application on the fly, without stopping, recompiling or rerunning. This approach is the most appropriate for applications with dynamic behavior. Finally, Section 6 summarizes and concludes our work.

2. Approaches to Automatic Performance Analysis

The first approach is a static automatic performance analysis. It is based on trace files and the analysis is carried out post-mortem. The main feature of this approach is that it introduces certain automatic techniques that analyze the trace file and that it provides hints to the non-expert user on the performance tuning phase. Carrying out an off-line analysis has the advantage that it does not introduce any extra overhead into the execution of the application, with the exception of the monitoring overhead. Since time is not so critical and the detailed information is available in the trace file, it is possible to perform the comprehensive analysis: identify the most important performance bottlenecks, determine their causes, relate them to the application source code and finally suggest certain recommendations to the user. The main disadvantages of such an approach are the following:

- To make the complete analysis, the trace file must contain detailed information about the application execution. This may require many events to be generated by instrumentation code. In some cases, such instrumentation can introduce a substantial intrusion in the execution of the application, or even hide its real behavior.

- The amount of data generated during tracing of a real distributed application can be substantial. The trace files can contain several gigabytes of data, making these files particularly difficult to store and process. However, it should be borne in mind that a very large application usually contains iterations of the same part, and, in stable conditions, information can be analyzed only once, since it is repeated many times in the trace.

- The analysis is based on a single application run, and the recommendations and tuning are adequate for that execution. Certain applications can change their behavior during different runs, or even along a single run. In these cases, the modifications suggested by the post-mortem analysis can be insufficient to cover the dynamic behavior of the application.

Taking these features into account, it can be deduced that this approach is suitable for applications that are not very large, having a fairly stable behavior in different runs and in the different iterations of the same run. In this case, the amount of data to be analyzed is not so large and the comprehensive analysis can provide useful recommendations for most of the application executions.

However, there are many applications that are too large to make this approach feasible, or the amount of data generated and the intrusion introduced due to the required instrumentation are too high. For these cases, our second approach, namely dynamic automatic performance analysis, may be more appropriate. In this approach, the application is monitored and diagnosed during its execution. The automatic analysis phase determines the most important problems, correlates them with application processes, modules or functions and attempts to

explain their reasons to developers. This approach offers the following advantages:

- The analysis defines and processes the performance measurements at runtime. Trace files are no longer needed. The instrumentation can be added or removed automatically according to the actual program behavior. The instrumentation overhead can therefore be reduced and controlled.
- The on-line analysis can focus on specific execution aspects (i.e. most severe problems), selectively refining its measurements in light of the previous results. This leads to a reduction in the amount of measurement data.
- Problems can be identified significantly faster than in a post-mortem approach.
- Analysis can be executed on a separate machine without introducing any direct overhead to the execution of the application, except for the limited instrumentation and certain data transfer over the network.

The dynamic approach is best suited for iterative programs and can handle long-running applications with high data volumes. However, similarly to static analysis, the dynamic analysis is based on a single run of the application.

When the application behavior depends on the input data or on the iteration of the execution, the suggested recommendations can be inadequate for a further run of the application.

Therefore, when the behavior of the application is so variable, it is more appropriate to apply our third approach – a dynamic automatic performance tuning. In this approach, the application is tuned on the fly without stopping, recompiling, or rerunning it. Therefore, the developer can concentrate on the design and development phase and, after this, dynamic tuning takes control of the application by monitoring the execution, analyzing behavior and modifying the application to improve performance. When the application behavior depends on the input data set or varies from one iteration to another, the analyzer detects the current behavior and determines how to adapt the implementation to changing conditions for the present execution.

Performance tuning must be carried out at run-time; there are therefore certain points to be considered:

- The intrusion must be minimized. Besides classical instrumentation intrusion, in dynamic performance tuning there are certain additional overheads due to monitor communication, performance analysis and program modifications.
- The analysis must be quite simple, as decisions have to be taken in a short time to be effective in the execution of the program.
- The modifications must not involve a high degree of complexity, as it is not realistic to assume that any modification on any application in any environment can be done on the fly.

For all these reasons, evaluation and modifications cannot be complex. Since all the tuning must be done in execution time, it is difficult to carry this out with-

out previous knowledge of the structure and functionality of the application. As the programmer can develop any kind of program, the potential bottlenecks can therefore be complicated. In such a situation, the analysis and the modifications might be extremely difficult. If knowledge of the application is not available, the applicability and effectiveness of our approach might be significantly reduced. Therefore, an effective solution is to take advantage of application knowledge prepared by the developer or to automatically extract as much information as possible from the unknown application.

In the following sections, the three approaches investigated are presented in detail.

3. Static Automatic Performance Analysis

3.1 Objectives

In this approach the goal is to provide a static automatic performance analysis tool that allows the user to skip all the performance analysis steps. The user can concentrate on the application design phase and then receive a direct recommendation from the tool that highlights the main performance bottlenecks, indicates the causes and provides certain suggestions concerning the possible actions required to improve performance. The user then simply takes the recommendations and makes the suggested changes in the source code.

Basically, the objective of this methodology is to analyze the most important performance problems of a parallel/distributed application and to obtain a list of suggestions that will improve the performance of the application.

KAPPA-PI (Knowledge based Automatic Parallel Program Analyzer for Performance Improvement) [5, 8] is an automatic performance analysis tool based on the static analysis of post-mortem traces of parallel and distributed applications, written with message passing paradigm (PVM or MPI libraries).

The first step in carrying out the analysis is to execute the application with a tracer tool that captures all the events related to the message passing primitives that have occurred during the execution. Our tool uses the trace file (captured events) as input and a knowledge base of performance bottlenecks [9]. It then tries to identify behavioral patterns in the trace file and to carry out the required analysis to determine the real causes of the bottleneck. Finally, it suggests recommendations (hints) to the users.

In the following subsection, we show a basic design of our tool called Kappa-Pi 2.

3.2 Basic Design of Kappa-Pi 2

The first version of Kappa-Pi [5] was a research tool used as a proof of concept. Several points have been modified in the design and implementation

of Kappa-Pi 2. The most significant improvements in the new version of Kappa-Pi are related to:

- Performance Data Inputs: Application execution related data as collected events, and performance knowledge representation.
- Performance Analysis stages: pattern matching, evaluation, static source code analysis and suggested recommendations.

3.2.1 Performance Data Inputs. Kappa-Pi 2 uses an internal trace format, for representing collected events, which is a general trace format for message passing primitives. This implies that the format can represent any event produced during the execution of a message passing application. The required filters and converters have been implemented to transform TapePVM trace files and VampirTrace files to this internal format. In this way, the tool is completely independent from the input trace file and the only requirement is to write the adequate format converter.

The performance bottlenecks specification (knowledge base) included in the first version of Kappa-Pi has been extended according to the performance properties description from APART project [9, 13]. A specification language based on ASL (APART Specification Language), and codified in XML form, has been defined to describe the bottleneck catalog (performance knowledge). The tool core reads this specification from a file. The inclusion of new performance bottlenecks, therefore, does not modify the tool implementation. Figure 1 shows the specification of a "late sender" bottleneck.

```
<PATTERN Name="Late Sender">
<ROOTTYPE>RECV</ROOTTYPE>
        <INSTANTIATION>
                <EVENT NAME="S1" TYPE="SEND" TO="ROOT"></EVENT>
                <EVENT NAME="E1" TYPE="ENTER" FROM="ROOT"></EVENT>
                <EVENT NAME="E2" TYPE="ENTER" FROM="S1"></EVENT>
        </INSTANTIATION>
        <CONSTRAINT>
                <COND TYPE="=" OP1="E1.type" OP2="RECV"></COND>
                <COND TYPE="=" OP1="E2.type" OP2="SEND"></COND>
                <COND TYPE=">" OP1="E2.stamp" OP2="E1.stamp"></COND>
        </CONSTRAINT>
        <EXPORT>
                <COMPUTE NAME="idle_time" AS="-" OP1="E2.stamp" OP2="E1.stamp"></COMPUTE>
        </EXPORT>
</PATTERN>
```

Figure 1. Late Sender problem specification represented in XML.

From the initial specification of bottlenecks (patterns), a decision tree is built. Since the specification is codified in XML, it is simple to create such tree.

3.2.2 Performance Analysis Stages. Figure 2 shows the basic architecture of the Kappa-Pi 2 tool.

At an initial stage, the detection of a bottleneck can be viewed as a Pattern Matching process between the events included in the Trace file and the tree that describes the performance patterns included in the Problem specification. The detection is guided by the decision tree applying the inefficiency patterns from their root events. The patterns found are summarized in terms of location (code spatial presence) and repetitions (temporal presence).

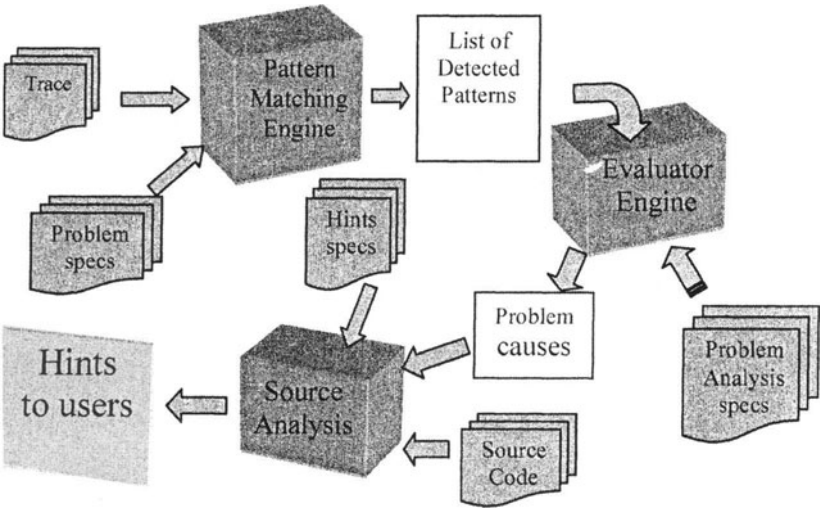


Figure 2. Kappa-Pi 2 basic architecture.

At a second stage, an Evaluator that takes the Problem Analysis specification as input determines the patterns over the specified thresholds and identifies the possible causes of the problem. After the evaluation, a list of deduced causes for the present patterns is obtained.

At a third stage, these causes are put in correspondence with locations in the source code (line, file, module, etc.). Knowing the cause of the bottleneck and the source code involved, a detailed analysis of such code (Source Analysis) is carried out to build a recommendation for the user. This analysis is based on the Hint specification that describes the relations and dependencies involved in the problem. This process finally provides the user with a set of recommendations (Hints), an explanation of inefficiency causes and a series of code changes (or code structure changes) that can be made directly on the code to resolve the performance problem.

For example, in a context of Master/worker application, the analysis can provide a hint as: "Synchronous Master/Slave situation found, master process

in main.c file while the slaves are in slave.c, Bored Master Problem, the ideal number of slaves is 3, change configuration of pvm_spawn in line 56 in main.c file."

4. Dynamic Automatic Performance Analysis

4.1 Objectives

Our goal is the creation of a tool that is able to analyze the performance of parallel applications during their execution, automatically detect bottlenecks and explain their reasons to developers. We must be able to provide a clear explanation of the identified problems, that is, we have to indicate the most important bottlenecks, explain their reasons, correlate them with the source code and, in addition, provide recommendations on possible solutions.

Ease of use is a important aspect for any tool. Therefore, we want to perform the analysis of application programs in a fully automatic manner and avoid the need for their manual instrumentation. It would be desirable to simply execute an existing program or even attach to one that is already running and to analyze it on-the-fly. To achieve these goals, it is necessary to apply a dynamic instrumentation technique [4, 10, 11, 12]. Moreover, we pay careful attention to intrusion issues and try to limit the amount of measured and processed data.

As knowledge of performance problems evolves, we cannot limit our tool to a few predefined and hard-coded rules. Therefore, we express our performance problem catalog in a declarative manner using APART Specification Language (ASL) [9, 13]. The tool provides its catalog and during the analysis process, must be able to interpret and evaluate the declared knowledge. Such an approach is flexible and extensible so that expert users can customize or extend the catalog to their specific requirements.

We target our tool at message passing parallel applications. However, we have tried to keep the open design and modular implementation to facilitate adaptations to different platforms or programming paradigms.

4.2 Basic Design

To achieve the objectives set out by our work, we combine dynamic performance monitoring with automatic on-the-fly analysis based on declarative problem specification. Essentially, the tool requires the following elements:

- a performance data collection mechanism
- a performance problem catalog
- a performance analysis process

In the following subsections, we present the functions of all the elements and justify their presence.

4.2.1 Performance Data Collection. A performance bottleneck search requires detailed information about program structure and its dynamic behavior during execution. This includes static information such as program modules, libraries, functions, call graph, and dynamic information such as calls statistics, timing statistics, communication patterns, resource usage. To extract this information, we use DynInst API [14].

Each of the application processes is executed without any changes under control of the dynamic monitors. The static information is gathered dynamically at application startup by parsing executables. During execution, when the analyzer requests certain specific performance measurements, a monitor automatically generates and inserts the instrumentation into the process and starts collecting the data. Later, when the information is no longer needed, the instrumentation is removed. The instrumentation acts at a function level and supports basic primitives such as timing and counting statistics. Additionally, the monitors gather low-level operating system statistics to provide constant general overview of the application performance (including for example metrics such as CPU-time, I/O time, or memory usage). This is performed by means of kernel microstate accounting [15], without imposing high overhead. The collected statistical data is periodically transmitted from the monitors to the analyzer process.

4.2.2 Performance Problem Catalog. To describe the performance bottlenecks, we use the ASL language. ASL is a declarative specification language that uses high-level abstractions called performance properties to represent common performance problems. Properties describe the specific types of performance behavior in a program. These are based on conditions dependent upon certain performance metrics. The existence of properties is associated with some level of confidence and with severity that estimates their importance. The performance analysis is based on the detection and evaluation of existing properties. The most severe properties represent performance problems.

The application and its performance data is represented by means of an object model. The model contains all static application entities, called regions, such as processes, modules and functions, as well as their relations and references to the source code. Regions can be used to guide the search for the location of potential problems. During execution, each application region can have its dynamic execution profile. The profile contains a set of predefined performance metrics that can be measured on demand (i.e. CPU time spent in region, number of calls). The performance properties always refer to existing model regions and metrics. An example performance property is presented in Figure 3. This property represents the I/O bottleneck caused by a high number of small disk requests in a given Region (e.g. function) of the application.

```
property small_io_requests (Region r, Experiment e, Region basis)
{
  let
     float cost = profile (r,e).io_time;
     int num_reads = profile (r,e).num_reads;
     int bytes_read = profile (r,e).bytes_read;
  in
  condition : cost > 0 and
        bytes_read/num_reads < SMALL_IO_THRESHOLD;
     confidence: 1;
  severity : cost/duration (basis, e);
}
```

Figure 3. Too small I/O requests problem, represented as performance property.

The object model exposed by the analyzer is limited by the capabilities of the monitoring processes. If certain metrics are not supported, the properties that use them cannot be evaluated.

The evaluation of all performance properties can result in the high cost of measurements. The key issue is to decide what properties should be evaluated, and in what order, so that intrusion can be reduced. The solution is to exploit generalization-specialization dependencies between properties. For example the late sender property should be evaluated only if the communication cost property holds. Therefore, we extend the ASL notation with a concept of hierarchical properties. The hierarchies can guide the automatic search process and limit the number of properties that must be evaluated.

In our catalog, the top-level properties refer to CPU-usage, communication cost, I/O cost and synchronization cost. Each branch is then expanded with more specific properties. For example the communication cost is divided into point-to-point and collective-communication cost. The hierarchy ends with low-level leaf-properties such as small_messages or high_network_contention property.

4.2.3 Performance Analysis Process. The dynamic performance analysis is a cyclic process based on a top-down approach. Figure 4 presents the basic design of the system that carries out such an analysis.

The analysis starts with the selection of top-level properties for all the application processes. The selected properties are first pre-evaluated to determine the required performance metrics. Next, the measurement requests are sent to monitors and the necessary data is collected for a specific period of time. When the data becomes available, the application model is updated and the selected properties are evaluated. In the following step, the holding properties are ranked by their severity. The most severe is expanded and its sub-properties are selected for further evaluation. The process continues the top-down search until

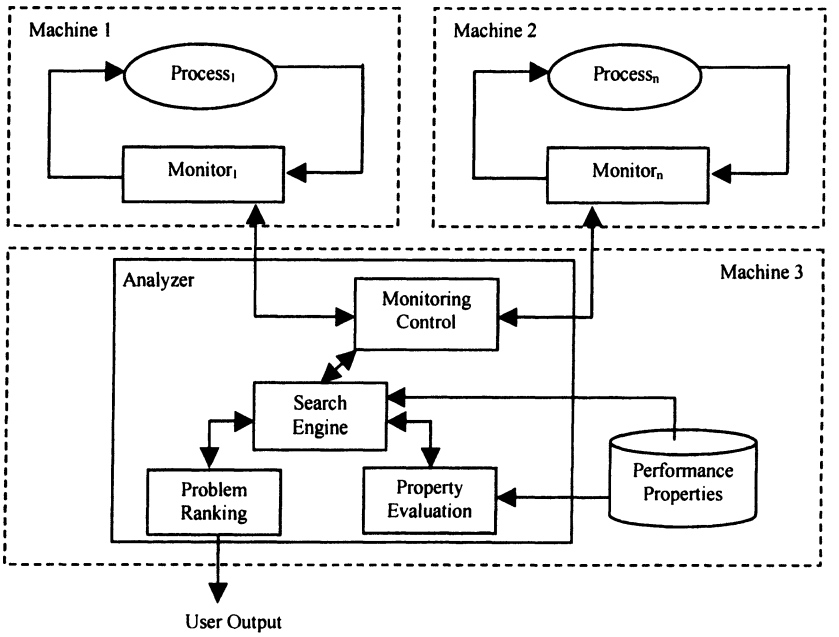


Figure 4. Basic design of the dynamic automatic performance analysis system.

reaching the most specific property (for example small_io_request as shown in Figure 3). In that case, the search can be concentrated on more specific process regions (i.e. functions) to detect a more precise problem location.

During the execution, the current property ranking is reported to the user. Each property is presented with its textual explanation, its context in the hierarchy and the location where the property is found (for example in a particular process or function). Additionally, for leaf-properties, the tool attempts to offer a possible solution to the problem.

5. Dynamic Automatic Performance Tuning

5.1 Objectives

The main goal of our dynamic performance tuning environment is to improve performance by modifying the program during its execution. To obtain better performance in such an approach, several steps are required: tracing of the application, analyzing performance behavior and modifying the running program. Moreover, all these steps must be performed automatically, dynamically and continuously during application execution.

The running parallel application is automatically traced, analyzed and tuned without the need to re-compile, re-link and restart. To accomplish this objective,

it was necessary to use dynamic instrumentation techniques [10] that allow the modification of application executable code on the fly.

Since parallel application consists of several intercommunicating processes physically executed on different machines, it is not enough to improve task performance separately without considering the global application view. To improve the performance of the entire application, we need to access global information about all associated tasks on all machines. To achieve this goal, we need to distribute the modules of our dynamic tuning tool to machines where application processes are running.

An issue of particular importance is the representation of knowledge that we can utilize when optimizing an application (i.e., measure points, performance model and tuning points, as explained later). This knowledge should be specified independently from the tool implementation, to permit extensibility and the inclusion of new performance problems. It would be ideal to use external files that could provide all the required information about the application.

5.2 Basic Design

Our dynamic performance tuning environment is implemented for PVM-based applications [16]. To make changes during the application execution, we use a dynamic instrumentation library called DynInst [14]. A dynamic tuning environment consists of three main components that cooperate amongst themselves, controlling the execution of the application (see Figure 5):

- set of distributed Tracer modules
- global Analyzer module
- set of distributed Tuner modules

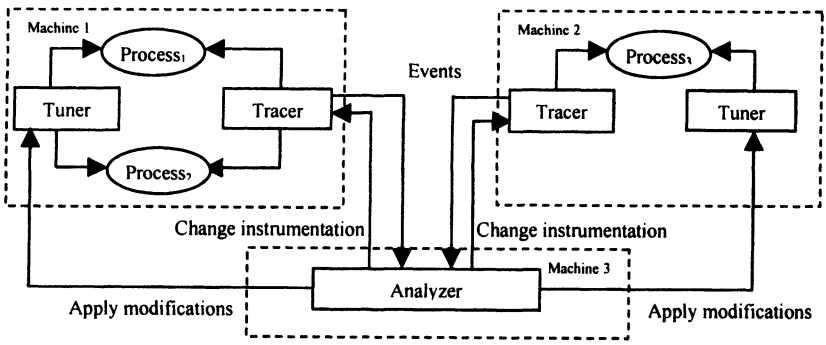


Figure 5. Dynamic performance tuning.

In the following subsections, we describe the components of our environment in detail.

5.2.1 Tracers. The tracing modules (Tracers) control all the processes of the parallel application and collect the events produced during the execution. To control all processes, Tracer modules are distributed to all machines where application processes are running. Each Tracer instance is responsible for controlling one machine. The distribution of the Tracer processes implies the need to control Tracer modules; we therefore distinguish between one master Tracer process and many slave Tracer processes. To collect events, the Tracer dynamically inserts instrumentation into the original program execution. Each event contains information about what happened, when and where. Because our approach assumes the global view of the application, we have to gather all events at a central location. Therefore, when an event (or a set of selected events) is generated, it is sent to the Analyzer module. Generally, the instrumentation points (measure points that represent what should be collected from the execution) can be specified by a user, before program execution. Instrumentation points can also vary on demand during run-time. If the Analyzer requires more or less information, it can notify the Tracer to change the instrumentation dynamically. Consequently, the Tracer must be able to modify the set of collected events i.e., to add more or to remove redundant instrumentation.

5.2.2 Analyzer. The analysis module (Analyzer) is responsible for the automatic performance analysis of a parallel application on-the-fly. It continuously receives events generated by different processes during the execution. By examining the set of events that enters into a time window, the Analyzer detects performance bottlenecks, determines the causes and decides what should be tuned in the application program. The Analyzer uses application knowledge (performance model) of possible problems and their solutions. When this module encounters the solution, it sends a request to the appropriate instance of Tuner, determining what should be changed, and where. For example, if in a particular master-slave application, the Analyzer determines that a work size parameter should be changed in a master process, the name of a variable, together with its new value, is sent to the Tuner. Obviously, during the analysis, Tracer modules are collecting and providing new data to the Analyzer. In certain cases, the Analyzer may need more information about program execution to determine the causes of a particular problem. It can therefore request the Tracer to change the instrumentation dynamically, depending on the necessity to detect performance problems. The analysis may be time-consuming, and can significantly increase the application execution time if both - the analysis and the application - are running on the same machine. To reduce intrusion, the analysis should be executed on a dedicated and distinct machine. Detected problems and recommended solutions are also reported to users.

5.2.3 Tuners. The tuning modules (Tuners) automatically change the application execution by inserting modifications into the running processes. Tuners manipulate the process image in memory, hence they have no need to access a source code or restart the application. Similarly to the Tracer modules, the Tuners must have access to all application processes. Therefore, the Tuner modules are distributed among all the machines where the application is running. A Tuner module waits for requests from the Analyzer. When a problem has been detected and the solution has been given by the Analyzer, the Tuner must apply modifications dynamically to the appropriate process or processes on the machine where it is running. All modifications are performed on tuning points. We consider several tuning techniques, such as modifying the value of a particular parameter in the application, replacing function calls with calls to different implementations, inserting additional code and others.

The changes made by Tuner will be invoked the next time the application reaches that point. The methodology can only be applied to problems that appear several times during the execution of the application. This fact might appear to be a constraint. However, it must be pointed out that the main performance problems of parallel/distributed application are those that appear many times during the execution of the application.

5.3 Dynamic Tuning Approaches

We consider two main approaches to dynamic tuning:
- the black box approach - automatic
- the cooperative approach - automated

In the automatic approach, an application is treated as a black box, because no application-specific knowledge is provided. This approach attempts to tune any application and does not require the developer to prepare it for the tuning (no changes are introduced to a source code). In this approach, the clue question is what information can be extracted from an unknown application. We focus on a set of problems related to the paradigm used to implement the application, as well as to the low-level functionality common to many applications. Consequently, we can dynamically optimize certain libraries that are used to develop the applications (e.g. PVM libraries, standard C-libraries).

For example, PVM-based applications use a specific communication mode - direct or indirect [16]. It is well known that, in most cases, the direct mode is more efficient than the indirect. When an application uses the indirect mode, the Analyzer can evaluate the conditions and determine whether the direct mode can be applied. If so, the Tuner can easily change the mode during run time. Another example is based on fact that the applications use general-purpose memory allocators (i.e. malloc). Standard allocators are known to be inefficient in particular cases such as the use of a large number of small objects.

In such conditions, it is highly reasonable to use optimized, custom memory allocator. When an inefficient memory allocation usage pattern is detected, our tool chooses an appropriate allocator to replace the standard.

For each problem, we define what should be measured. We determine the general performance model and the appropriate solutions, basing our decisions on the experiments conducted. In this approach, both user participation and source code changes are eliminated.

In the cooperative approach (automated) we assume that an application is tunable and adaptable. This means that the developers must prepare an application for the possible changes. They must specify a knowledge that describes what should be measured in the application (measure points), what performance model should be used to evaluate the performance, and finally what can be changed to obtain better performance (tuning points).

The possible examples include the determination and tuning of optimal work size for a master/worker application (parameter tuning), optimal number of running workers (parameter tuning), selection of the most efficient strategy of data distribution (implementation selection).

6. Conclusions

Performance analysis and the tuning of distributed applications are key issues in the design and development of such applications. However, this performance analysis is a complex task that requires a high degree of expertise. Therefore, automatic performance analysis tools are required to help and guide the user/developer in the application tuning phase. In this paper, three different approaches have been described. Each approach has target applications, with specific features, and is intended to satisfy specific user needs, according to user capabilities. The first approach is the static automatic performance analysis. In this approach there is detailed information available, and the complete analysis is carried out in a post-mortem phase by considering all such information. This approach is suitable for applications that do not present variable or dynamic behavior depending on the input data set. The second approach is the dynamic automatic performance analysis. This avoids trace files and allows for controlling the amount of instrumentation inserted in the application; it is suitable for large applications with a stable behavior. The third approach is the dynamic automatic performance tuning, which tries to adapt the application to existing conditions on the fly, without stopping, recompiling or rerunning it. This approach is the most appropriate for applications having dynamic behavior.

Acknowledgments

This work has been supported by the MCyT (Spain) under contract TIC2001-2592, by the European Commission under contract IST-2000-28077 (APART),

and partially supported by the Generalitat de Catalunya - GRC 2001SGR-00218.

References

[1] Heath, M.T., Etheridge, J.A. Visualizing the performance of parallel programs. *IEEE Computer*, 28:21-28, November 1995.

[2] Nagel, W.E., Arnold, A., Weber, M., Hoppe, H.C., Solchenbach, K. Vampir: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69-80, 1996.

[3] Reed, D.A., Roth, P.C., Aydt, R.A., Shields, K.A., Tavera, L.F., Noe, R.J., Schwartz, B.W. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceeding of Scalable Parallel Libraries Conference*, pp. 104-113, IEEE Computer Society, 1993.

[4] Miller, B.P., Callaghan, M.D., Cargille, J.M. Hollingswoth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam K., Newhall, T. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28:37-46, November 1995.

[5] Espinosa, A., Margalef, T., Luque, E. Relating the Execution Behaviour with the Structure of the Application. In *EuroPVM/MPI 1999, LNCS*, 1697:91-98, 1999.

[6] Wolf, F., Mohr, B., Automatic Performance Analysis of MPI Applications Based on Event Traces. In *EuroPar 2000, LNCS*, 1900:123-132, 2000.

[7] Truong, H.L., Fahringer, T. Scalea: A Performance Analysis Tool for Distributed and Parallel Programs. In *EuroPar 2002, LNCS*, 2400:75-85, 2002.

[8] Espinosa, A., Margalef, T., Luque, E. Automatic Performance Analysis of PVM applications. In *EuroPVM/MPI 2000, LNCS*, 1908:47-55, 2000.

[9] Fahringer, T., Gerndt, M., Riley, G., Larsson, J. Specification of Performance problems in MPI Programs with ASL. In *Proceedings of ICPP*, pp. 51-58, 2000.

[10] Buck, B., Hollingsworth, J.K. An API for Runtime Code Patching. University of Maryland, Computer Science Department, *Journal of High Performance Computing Applications*, 2000.

[11] Paradyn Project. *Paradyn Parallel Performance Tools, User's Guide, Release 3.3.* University of Wisconsin, Computer Science Department, January 2002.

[12] Cain, H.W., Miller, B.P., Wylie, B.J. A Callgraph-Based Search Strategy for Automated Performance Diagnosis. In *Euro-Par 2000, LNCS*, 1900:108-122, 2000.

[13] Fahringer, T., Gerndt, M., Riley, G., Traff, J.L. *Knowledge Specification for Automatic Performance Analysis*, Tech. report, FZJ-ZAM-IB-2001-08, 2001.

[14] Hollingsworth, J.K., Buck, B. *DyninstAPI Programmer's Guide, Release 3.0.* University of Maryland, January 2002.

[15] Cockcroft, A. *Sun Performance and Tuning: Sparc and Solaris (2nd Edition)*, Sun Microsystems Press, October 1994.

[16] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V. *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing.* MIT Press, Cambridge, 1994.

PERFORMANCE MODELING OF DETERMINISTIC TRANSPORT COMPUTATIONS

Darren J. Kerbyson
Performance and Architecture Laboratory, CCS-3
Los Alamos National Laboratory, Los Alamos, NM, USA
djk@lanl.gov

Adolfy Hoisie
Performance and Architecture Laboratory, CCS-3
Los Alamos National Laboratory, Los Alamos, NM, USA
hoisie@lanl.gov

Shawn D. Pautz
Transport Methods, CCS-4
Los Alamos National Laboratory, Los Alamos, NM, USA
pautz@lanl.gov

Abstract In this work we present a performance model that encompasses the key characteristics of a Sn transport application using unstructured meshes. Sn transport is an important part of the ASCI workload. This builds on previous analysis which has been done for the case of structured meshes. The performance modeling of an unstructured grid application presents a number of complexities and subtleties that do not arise for structured grids. The resulting analytical model is parametric using basic system performance characteristics (latency, bandwidth, MFLOPS rate etc), and application characteristics (mesh size etc). It is validated on a large HP AlphaServer system showing high accuracy. The model compares favorably to a trace based modeling approach which is specific to a single mesh/processor mapping situation. The model is used to give insight into the achievable performance on possible future processing systems containing thousands of processors.

Keywords: performance evaluation, performance modeling, unstructured meshes, large-scale systems

1. Introduction

In this work we present the development and use of a performance model for an application code developed for solving the Boltzmann equation deterministically (Sn transport) on unstructured meshes. The unstructured mesh application that is modeled here is known as Tycho [12]. The problem is of great importance to the ASCI workload, hence other codes are also being under development for this purpose e.g. [14].

Unstructured meshes have several benefits over the use of structured meshes in terms of the calculations undertaken, but have significant extra overhead in terms of performance. Several important performance factors that can reduce the overall calculation efficiency of this type of computations on large-scale parallel systems are analyzed in this paper.

The algorithms employed in deterministic Sn (discrete ordinate) computations fall in a class generically named wavefront techniques. In a nutshell they utilize an iterative approach using a method of "sweeping" [3]. Each spatial cell in a mesh is processed in a specified order for each direction in the discrete ordinates set. The wavefronts (or sweeps) are software pipelined in each of the processing directions. Wavefront algorithms exhibit several interesting performance characteristics, related to the pipelined nature of the wavefront dynamics. These include a pipeline delay across processors for a sweep, and a repetition rate of both computation and communication in the direction of the sweep. In the case of a structured mesh a high efficiency of calculation can be achieved as all active processors perform the same amount of work, and communicate the same sized boundary data [3].

Efforts devoted to the performance analysis of Sn transport date back many years. Research has included the development of performance models as a function of problem mesh and machine size [8]. More detailed performance models have been developed that also include inter-processor communication, and SMP cluster characteristics [3–4]. However, these all considered an underlying structured mesh.

The key contribution of this paper is the development of an analytical performance model of Sn transport on unstructured meshes. This is the first performance model for Sn Transport on unstructured meshes. The model encapsulates the main performance characteristics parameterized in terms of mesh size and system configuration. Two analytical models are considered, one in a general form and one in a mesh specific form. A third "trace" model, similar to Dimemas [2], is included to compare the different models.

The approach that we take for modeling is application centric. It involves understanding the processing flow in the application, the key data structures, and how they use and are mapped to the available resources. From this a performance model is constructed that encapsulates the key performance char-

acteristics. This approach has been successfully used on an adaptive mesh code [6], a structured mesh transport code [3], and a Monte-Carlo particle simulation code [9].

The analytical model developed here is shown to have reasonable accuracy through a validation process on a HP AlphaServer parallel machine. The general model is able to add insight into the achievable performance that could be obtained on hypothetical future architectures and hence indicating the efficiency and sizes of mesh that could be processed. Specifically we use the model to explore expected achievable performance on future large-scale systems which may be capable of a 100 tera-flops prior to their availability.

The paper is organized as follows. In Section 2, the Sn transport calculation is detailed and comparisons between its operation on structured and un-structured meshes are made. In Section 3 the key characteristics of the processing are described which are used in the development of the performance models in Section 4. The models are validated in Section 5 on a number of unstructured meshes, and are used to explore the performance on future architectures that cannot currently be measured in Section 6.

2. Overview of Sn Transport Algorithms

2.1 The method of sweeping

Two examples are depicted below that illustrate the method of sweeping used within an Sn transport calculation for a structured and an unstructured mesh. In both cases, the calculation dependencies in the direction of the sweeps are clearly shown.

Structured meshes In three-dimensions, each sweep direction can be considered to originate in one of the 8 corners ("octants") of the spatial domain. Within each octant, the ordering of cell processing is identical. Figure 1 shows the first six steps of two separate sweeps at different angles for a two-dimensional spatial domain, originating from different octants. The edge of the sweep corresponds to a wavefront and is shown as black. It requires the grey cells to have been processed in previous steps. The same operation can take place in three dimensions resulting in a wavefront surface. The wavefront propagates across the spatial domain at a constant calculation velocity since the time needed to process a cell is constant. This processing algorithm as developed in [8], uses direct indexing of the spatial mesh as the cell processing order is deterministic for each sweep direction.

Unstructured meshes An example two-dimensional unstructured mesh is depicted in Figure 2. Two sweep directions are again used to illustrate the processing over a total of six steps. As before, the cells being processed in the

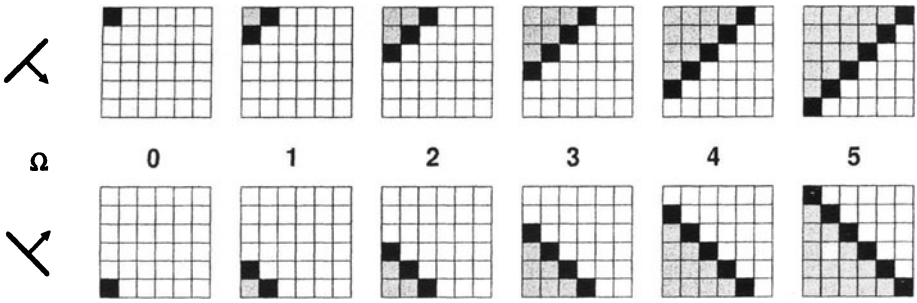


Figure 1. Example sweep processing on a 2-dimensional structured mesh.

current step are shown as black and require the previously calculated grey cells. The ordering of cell processing is direction dependent. The incoming data to a cell are determined by the mesh geometry, and it is apparent that the propagation speed of the wavefronts also varies with direction. The same situation occurs in three-dimensional geometry, only with the mesh being composed of tetrahedrons, pyramids, or prisms.

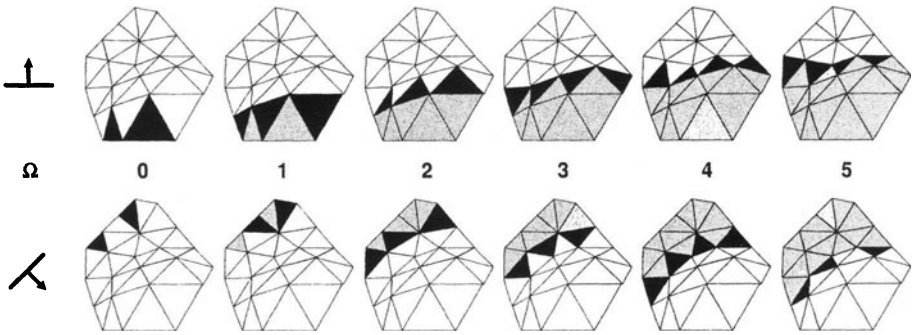


Figure 2. Example sweep processing on an 2-dimensional unstructured mesh.

2.2 Sweeping in Parallel

Parallel wavefront computations exhibit a balance between processor efficiency and communication cost [3]. Faster wavefronts, generated by a data decomposition leading to small subgrid sizes per processor, introduce higher communication costs but result in high processor utilization. The opposite holds true for slower moving sweeps due to larger subgrid sizes. In order to optimize wavefront dynamics, Sn transport applications typically utilize blocking of the spatial subgrid and/or blocking of the angle set.

Important performance considerations in parallel wavefront applications, which need to be captured in the model, are as follows:

pipeline effects a processor is inactive until a sweep surface enters cells in that particular processor. However, multiple sweeps are active at any given time in the processor array. Overlap exists between computation and communication within each sweep, and across the active sweeps.

communication costs for boundary data transfer.

load balancing of the number of cells processed on each PE in a step. This applies to wavefronts on unstructured meshes only as an equal number of cells are processed in each step on each PE for a structured mesh.

In order to analyze these effects, the processing that takes place on both structured and unstructured meshes is illustrated below.

Structured meshes In codes such as Sweep3D which performs an Sn transport computation on a structured meshes, the 3-D mesh is mapped onto a 2-D processor array such that each processor has a column of data which is further blocked in its third dimension. The processing is effectively synchronized after the first sweep has moved across the PE array resulting in all processors being active. The processing involved in each sweep is dependent on the block size (a known constant). Thus one diagonal of processors will be processing one sweep while the previous diagonal is processing the next sweep and so on (Figure 3). The direction of sweep travel is the indicated by Ω with inter-processor communications shown by arrows. It has been shown that the cost of performing this calculation on a structured mesh conforms to a pipeline model [3]:

$$T_{Total} = (P_x + P_y - 1)(T_{CPU} + ST_{msg}) + (N_{sweep} - 1)(T_{CPU} + 4T_{msg}) \quad (1)$$

where P_x and P_y are dimensions of the processor grid, Nsweep is the number of sweeps, T_{CPU} and T_{MSG} is the time to process a cell block, and the time to communicate a message respectively. The first part of this equation corresponds to the length of the pipeline and the second part it the number of repetitions once the pipeline is filled.

Unstructured meshes The processing on an unstructured grid follows the same dependency rules as above, but the mesh partitioning is typically done in all 3 dimensions. An example 2-D partitioning of an unstructured mesh is shown in Figure 4. The communications between processors are shown by arrows, and a simplified propagation of the sweep in the indicated direction is shown by the grey lines. Tycho actually enables sweeps in all directions to

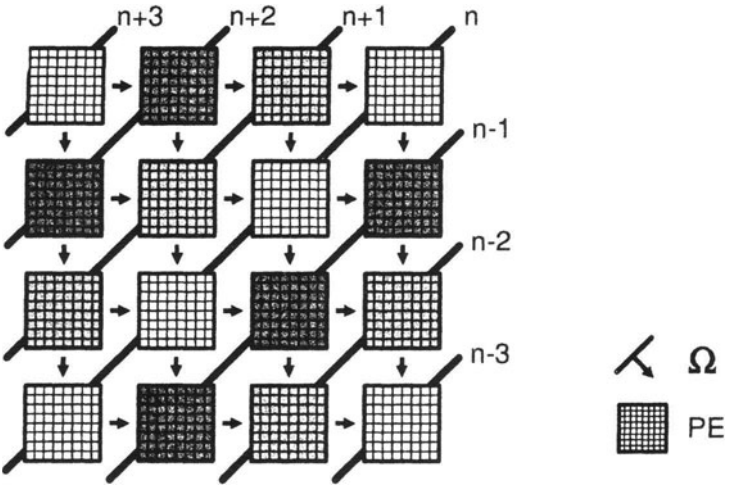


Figure 3. The pipeline processing of sweeps in parallel.

commence simultaneously. Thus each cell is processed for each sweep angle set whilst still taking into account the dependencies in each of the sweep directions. The unit of processing work can be considered as a single cell-angle pair.

The sweep processing on the unstructured mesh can also be blocked - a number of cell-angle pairs can be processed per step. However, this can result in processor inefficiency down the pipeline - processor idleness can occur due to boundary data between processors not being a constant size over steps.

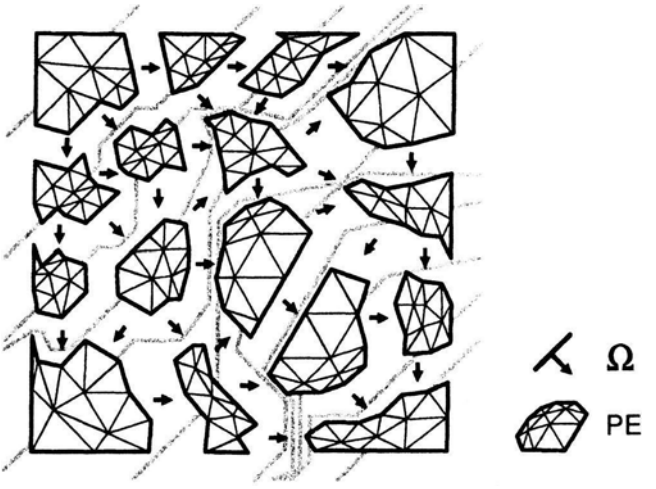


Figure 4. Example partitioning and sweep flow on a 2-D unstructured mesh.

An approach to maximize processor utilization is to consider each cell-angle pair as a task, and to assign each a priority. Tasks are placed in a priority queue with the highest priority ones being processed first. The key to this approach is in the assignment of priorities. In general, processor boundary cells will be of high priority (boundary cells need to be processed in order for the sweeps to travel down the pipeline), and cells upstream of the boundary cells (i.e. those needing to be processed prior to the boundary cells) are given an even higher priority. This scheduling approach attempts to maximize cell boundary production in order to keep the processors downstream in the pipeline busy. Several different heuristic scheduling schemes have already been analyzed using Tycho [12].

3. Key Processing Characteristics

The key processing characteristics in Tycho are: mesh partitioning, pipeline processing, processor utilization, and strong scaling. An understanding of these factors is required in order to formulate the performance model. Two situations are considered in this analysis - a general case when no knowledge of the mesh exists (except for the total number of cells), and a mesh specific case in which the mesh is inspected for detailed information which can be used in the model.

3.1 Mesh partitioning

The partitioning is not done within Tycho, rather a suitable tool such as Metis [5]is utilized. This mesh partitioner aims to produce equally sized partitions while minimizing boundaries. Such an optimal partitioning of the mesh in general would keep the work across PEs constant and minimize the communication cost. However, due to the pipeline processing and load-balancing characteristics in Tycho, this partitioning may not be optimal.

Tycho also utilizes a 3-dimensional processor decomposition. This contrasts with Sn transport on structures meshes which typically utilizes a 2-dimensional processor decomposition.

For such a 3-D partitioning, the number of cells per partition can be taken to be $E_p = N/P$ where N is the number of cells in the mesh, and P is the number of PEs (which is equal to the number of partitions). In the general case each 3-D partition would ideally have six nearest neighbors each with a boundary size of $Ep^{2/3}$ cells.

When considering a specific mesh the number of cells per processor, E_p, and a vector of neighbor communications, $N_c(s,p)$, and average communication sizes, $N_s(s,p)$, per step for each PE can be obtained by inspection once the mesh has been partitioned and the work scheduled. Note that the vectors are defined for each step, s, and for each PE, p.

3.2 Pipeline processing

Sweeps in all directions start simultaneously in Tycho. The first cell- angle pairs processed are those that lie on the boundary of the spatial mesh which have no inflows in the sweep direction. This corresponds to nearly all boundary elements. The sweeps thus generally start from the surface of the mesh and work their way to the centre before propagating out the opposite side. The dynamics of the pipeline is determined by the pipeline length and by the amount of computation done on each mesh partition. The pipeline length is determined by the number of stages in the propagation of the sweep from one side of the mesh to another. In 2-D the number of gray lines in Figure 4 would represent the number of stages. In general, given an ideal 3D partitioning, the pipeline length is given by:

$$P_L = (P_x - 1) + (P_y - 1) + (P_z - 1) \tag{2}$$

where P_x, P_y and P_z are the number of PEs in each of the three dimensions respectively. The total work done, or the total number of cell-angle pairs processed, on each mesh partition in an iteration is equal to:

$$W_p = E_p * N_\Omega \tag{3}$$

where N_Ω is number of sweep directions. For a specific mesh, the pipeline length, P_L can be obtained by inspection of the mesh after the partitioning has been performed and is equal to the maximum number of PEs traversed in any sweep direction. The total amount of work done per partition remains as above.

3.3 Processor utilization

Each step in Tycho consists of three stages: do the work at the top of the priority queue, send boundary data to PEs downstream, and receive boundary data from upstream PEs. The amount of work done in a step is determined by an input parameter $MCPS$ (MaxCellsPerStep) and specifies the maximum number of cell-angle pairs that can be processed in a step in each processor. Thus $MCPS$ effectively represents a blocking factor.

The processing situation is complicated by the processing dependence between upstream and downstream cells in the sweep directions. This dependence may lead to downstream PEs waiting for the upstream PEs to send the necessary boundary information. There will almost always be a degree of inefficiency in this operation and processors will be starved of work waiting for the results from other PEs. It is interesting to note that for the case of structured meshes, the work on each PE is equal throughout and thus there processors are fully utilized once the pipeline is filled. To quantify this inefficiency, the metric of Parallel Computational Efficiency, PCE [12], is used:

$$PCE = \frac{W_p}{\sum_{i=1}^{\#steps} \max_p(\|work(P,S)\|)} \tag{4}$$

where $work(P,S)$ is the number of cell-angle pairs processed in step S on processor P, and W_p is the total number of cell-angle pairs processed on each processor in an iteration. PCE represents the fraction of the maximum number of cells that are processed in all steps in an iteration. When $PCE = 1$ the efficiency is 100% - this can only occur on a small processor run (typically < 9 PEs). The lower the value of PCE, the greater the inefficiency.

A value for PCE can be obtained for a specific mesh after its partitioning and before the sweep execution. The number of steps required to perform the total number of cell-angle pairs per PE (excluding the pipeline effect) is given by:

$$\frac{W_p}{(MCPS * PCE)} \tag{5}$$

In the general case, i.e. without the inspection of the mesh, a value of the PCE has to be assumed - possibly based on experience from prior meshes. This assumption can be inaccurate reflecting the tradeoff between generality and accuracy always present in performance modeling work.

3.4 Strong Scaling

Typical Tycho runs are executed in a strong scaling mode - the input mesh size is constant and thus partitions become smaller on larger processor counts. This is easily incorporated into a modified expression for Wp. However, for the case of strong scaling, the memory hierarchy effects have to be carefully considered. For instance when a mesh partition becomes small enough to fit in cache the performance will be better than if main memory has to be accessed.

Figure 5 shows the computation time per cell for different meshes and partition sizes on an 833MHz Alpha EV68 processor with 8MB L2 cache. There are clearly three regions evident: when the partition does not fit into L2 cache (right hand plateau), when the mesh fits into L2 cache (left hand plateau), and when partial cache re-use occurs (middle region). There is some variation between meshes in this analysis due to the different memory access patterns and hence the actual cache reuse. It can be seen that a good approximation to this memory hierarchy performance can be encapsulated in a piece-wise linear curve. In general however, we are interested in large meshes - those that unfortunately will not exhibit cache re-use.

4. Performance Models for Tycho

Three performance models for Tycho are described below using the key characteristics described in Section 3. The first two, the General Model (GM) and

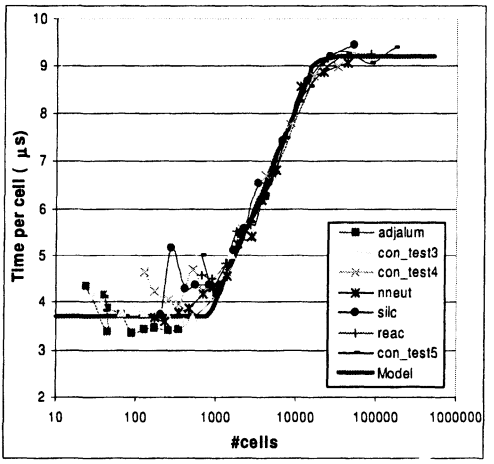


Figure 5. Processing time per cell on different mesh and partition sizes.

Mesh Specific Model (MSM) are analytic models. The GM uses the characteristics discussed in Section 3 for a general mesh - i.e. without using detailed knowledge on the mesh partitioning. The MSM on the other hand uses the knowledge of the partitioned mesh. These first two models are described together in Section 4.1 below. The third is a Trace Model (TM) that is based on an analysis of a communication trace obtained at run-time for a specific mesh on a specific processor count. The TM is described in Section 4.2.

4.1 Analytical performance models

In the analytical performance models we assume that the three stages of a Tycho step are distinct and do not overlap - those of computation, blocking sends, and blocking receives. This is a simplification as there will be a degree of overlap between computation and communication. However, the amount of overlap is assumed to be small, and as will be seen from the error analysis in Section 5, is a reasonable assumption. The runtime for an iteration of Tycho can be modeled as:

$$T_{iter} = \left(\sum_{S=1}^{\#steps} \max_{P}(Work(P,S)) \right) . T_{Elem}(\frac{N}{P}) + \sum_{S=1}^{\#steps} \max_{P} \left(\sum_{C=1}^{\|N_c(S,P)\|} T_{comm}(N_c(S,P,C), N_s(S,P,C)) \right) \quad (6)$$

where the first term represents computation and the second term represents communication. The model parameters are as follows:

$\#steps$	is the number of steps in an iteration
$Work(P,S)$	number of cell-angle pairs processed on processor P in step S
$N_c(S,P,C)$	destination PE for communication C in step S on processor P
$N_s(S,P,C)$	is the size of communication C in step S on processor P
$\|\|N_c(S,P)\|\|$	number of communications from processor P in step S
$T_{Elem}(x)$	time to process a cell-angle pair given x cells mapped to a PE.
$T_{comm}(x,y)$	time to communicate a message of size y bytes to processor x

Assuming no overlap between communication and computation, as stated earlier, in equation 6 the first term represents computation time and the second communication time. Due to the wavefront nature of these algorithms, within one wavefront, overlap exists between the computation of the angle-cells. The wavefront will be ready to propagate downstream as soon as the work on the largest subgrid contained in the wavefront will be completed. Hence the "max" function contained in the first term of the equation. Similarly, the second term is a sum of the non- overlapped communication steps for all wavefronts.

The model as formulated in equation 6 represents both the MSM and GM. However many of the parameters in the model are substantially different between the two. The parameters of $T_{Elem}()$, and $T_{comm}()$ are hardware specific and remain the same. A two-parameter, piece-wise linear model for the communication is assumed which uses the Latency (L_c) and Bandwidth (B_c) of the network communication.

$$T_{comm}(D,S) = L_c(S,D) + S.\frac{1}{B_c(S,D)} \tag{7}$$

where L_c is the communication latency, B_c is the communication bandwidth, D is the message destination PE, and S is the message size.

In the MSM the parameters steps, $Work()$, $N_c()$, and $N_s()$ represent actual time histories of the work and communications done through all the steps in an iteration. These time histories are obtained by inspection after mesh partitioning and scheduling of the cell-angle pair tasks, prior to actual processing in Tycho. They are specific to both the mesh and the processor count. This type of model tends to reflect the static behavior of the code while parameterizing the main dynamic attributes. A similar approach was successfully taken in the modeling of an adaptive mesh code [6].

In the GM the assumptions presented in Section 3 for the processing characteristics can be used to simplify the model into a general form. In the GM, the number of steps is given by

$$\#steps = (\frac{E_P * N_\Omega}{MCPS * PCE}) + (P_x - 1) + (P_y - 1) + (P_z - 1) \tag{8}$$

where E_p is the number of cells per PE (assumed constant at N/P), $MCPS$ is the MaxCellPerStep (input parameter to Tycho), $N\Omega$ is the number of sweep

directions, and PCE is the Parallel Computational Efficiency as described earlier. P_x, P_y and P_z are the number of processors in the logical x, y, and z dimensions respectively, as described in section 3.1. The first part of this equation represents the number of work steps and the second part represents the pipeline length (which will in general be an underestimate).

The work on each PE in each step is assumed a constant:

$$Work(P,S) = \min(MCPS, E_p * N_\Omega) \tag{9}$$

The number of communications per step on each PE, $\|N_c(S,P)\| = 6$, and average communication sizes per step on each PE are also assumed constant, $N_s(S,P,C) = \min(E_p^{2/3}, MCPS2/3) * 40$. Note that each boundary cell communicated consists of 40 bytes of data.

The communication time is subject to a contention in the communication network. Our experience on using Tycho, and other codes on clusters of SMPs, is that the main contention occurs on the number of out-of-node communications that occur simultaneously. For example with the fat-tree network of the Quadrics network [13], the number of communications that collide in higher levels of the fat-tree is low due to dynamic routing. The contention is taken into account by a multiplicative constant on the communication time, T_{comm}, which represents the number of out-of-node simultaneous communications.

4.2 A trace model

Traces that are obtained at run-time can capture the full computation / communication interaction of an application but is specific to a mesh and to a particular processor count. The trace can be effectively re-played in order to give a prediction. Such a trace modeling approach is not new and has been used in tools such as Dimemas [2], and PACE [7]. The approach taken here is similar to that of Dimemas and provides a comparison with the analytical models.

The traces used here contain three event types: 1) the number of cell- angle pairs processed in a step, 2) the communication sends, and 3) the communication receives. The communication events include details on the source and destination PEs as well as the message sizes. However, it should be noted that no timing information is stored in the trace file.

Timing information is produced by a Trace Model Evaluator (TME) developed at Los Alamos. The TME allows different prediction sub- models to be used to predict: the time to process a cell-angle pair, the communication costs, and the communication contention. Different sub- models may be used to predict component times for different systems. The TME effectively replays the trace file whilst accounting for the expected time taken by each event, and also resolving communication dependencies and possible contention in the communication network [10]. It also determines when bi-directional traffic occurs

between any two nodes, which can also reduce the effective communication bandwidth. The TME is a detailed evaluation method with a potential high accuracy but unfortunately loses any generality in the model. The evaluation for a given input trace file is specific to a mesh and processor count. As it will be seen below, the analytical models compare favorably with this more specific trace model.

5. Performance Model Validation

The three performance models are validated in this section on a HP AlphaServer ES40 64 node system. Each node in this system consists of 4 Alpha EV68 processors running at 833MHz each with an 8MB L2 unified cache. The nodes are interconnected using the Quadrics QSnet high speed network with Elan3 switching technology. The details of this architecture are not described in detail here but a good overview of its performance characteristics can be found in [13]. The hardware parameters for this system are listed in Table 1.

Table 1. Hardware parameters for the HP AlphaServer ES40 validation system.

$T_{elem}(E_p)(\mu s)$		9.2	$E_p \geq 16Kcells$
		$1.8Ln(E_p) - 8.4$	$800 < E_P < 16K$
		3.7	$E_P \leq 800$
$L_c(S,D)(\mu s)$	intra-node ($D \leq 4$)	12.7	$S < 64bytes$
		12.8	$64 \leq S \leq 256$
		30.3	$256 < S \leq 8192$
		25.7	$S > 8192$
	inter-node ($D > 4$)	9.28	$S < 64bytes$
		9.00	$64 \leq S \leq 256$
		21.4	$S > 512$
$1/B_c(S,D)(ns)$	intra-node ($D \leq 4$)	0.0	$S < 64bytes$
		24.0	$64 \leq S \leq 256$
		9.0	$256 < S \leq 8192$
		3.2	$S > 8192$
	inter-node ($D > 4$)	0.0	$S < 64bytes$
		25.5	$64 \leq S \leq 512$
		13.7	$S > 512$

Four meshes are used in the validation as listed in Table 2. These represent small and medium sized meshes, resulting in small mesh partitions on the largest processor configuration considered.

Measurements and model predictions for each of the four meshes are shown in Figure 6. The application input parameter $MCPS$ was set at 512 in all cases.

Table 2. Meshes used in the validation.

Mesh	*#Cells*	*Description*
Nneut	43,012	Neutron well-logging tool and surrounding media
Silc	51,963	Computer Chip and packaging for radiation shielding
Reac	165,530	Reactor pressure vessel and surrounding cavity structures
Con_test5	168,356	Cube divided into approximately equal-sized elements

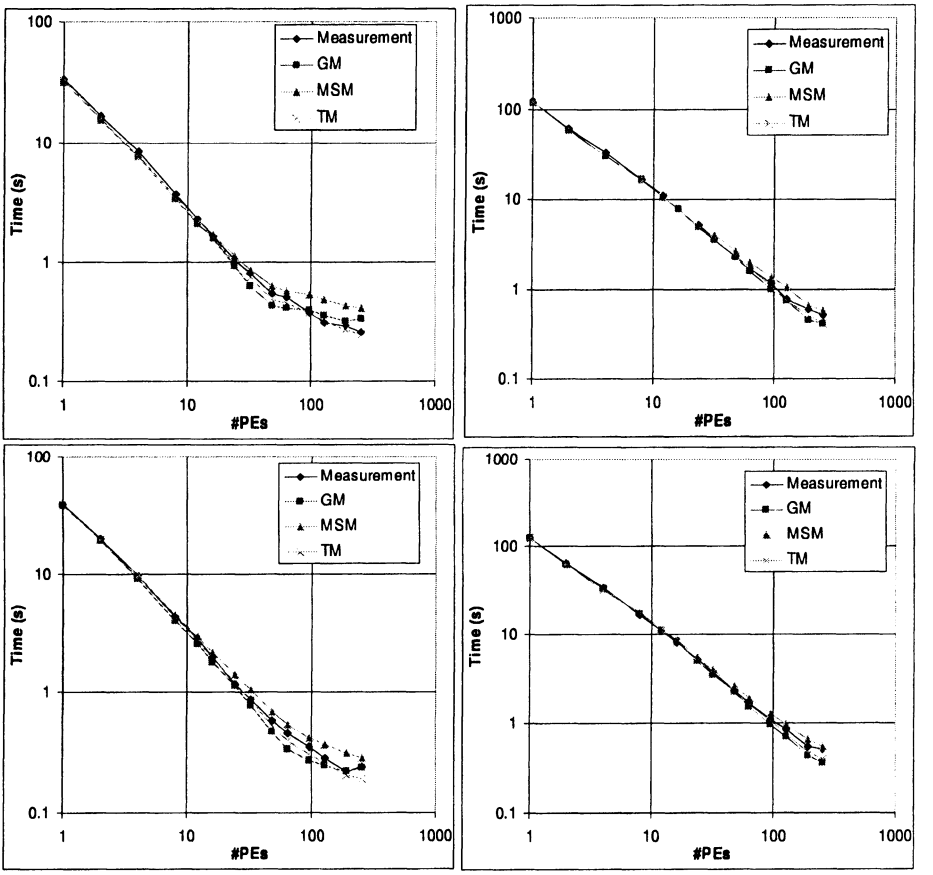


Figure 6. Model validation on a) Nneut, b) Silc, c) Reac, d) Con_test5.

A summary of the errors observed is listed in Table 3. It can be seen that all the models have a good prediction accuracy on small PE counts (case i., $P \leq 32$). This is expected since they all use the same model for computation, the dominant term in equation 6 for small processor counts. They do differ

however in their treatment of communication which becomes apparent on large processor counts (case ii., $P > 32$). The MSM will tend to over-predict since any possible overlap between computation and communication is not modeled. In contrast the GM will tend to under-predict since it assumes an idealized 3-D mesh having a minimum number of neighbors and hence a smaller degree of communication than actually occurs. The TM is the most accurate taking into account much of these effects, but requires a separate trace to be analyzed for each mesh/processor-count pairing.

Table 3. Summary of Model Prediction Errors (%).

	GM			*MSM*			*TM*		
	i.	ii.	Av.	i.	ii.	Av.	i.	ii.	Av.
Nneut	9.1	15.7	12.4	5.7	28.4	17.0	6.1	5.0	5.5
Silc	2.9	11.1	7.0	1.8	17.2	9.5	0.8	7.4	4.1
Reac	5.3	13.5	9.4	8.4	22.3	15.3	2.3	11.5	6.9
Con_test5	2.2	8.9	5.4	4.2	13.1	8.7	1.3	4.6	3.0
Overall			8.6			12.6			4.9

Given its general nature and its reasonable accuracy, GM is used in the following section to provide insight into the performance of Tycho on systems and configurations that can not be currently analyzed through measurements.

6. Performance Exploration

The performance models developed in Section 5 can be used in many different ways to explore the performance space of Tycho, on current and future system architectures. Here we use the validated GM model to: i) detail where time is spent on the current ES40 system (Figure 7). ii) analyze the impact on performance when using systems containing other processors while still using the Quadrics QsNet network (Figure 8). iii) predict the performance on larger meshes and system sizes (Figure 9). iv) optimize the runtime by calculating the value of MCPS that results in a communication cost constituting less than 20% of the total (Figure 10).

The performance characteristics of the processors used in these scenarios are listed in Table 4. These computational characteristics were based on measurement made on single processors, in a similar way to that depicted in Figure 5. Note that the Itanium with a 3MB L3 cache, and the EV68 1.25GHz with a 16MB L2 cache, have a different memory hierarchy than that of the validation system, so that the curve with the number of cells per PE E_p, is also different. All the parallel systems considered use the Quadrics network, whose

performance characteristics are unchanged from those listed in Table 1, with the processors listed in Table 4.

Table 4. Element processing time (Telem(Ep) in μs) on different processors.

Memory	Alpha EV68 1GHz	Alpha EV68 1.25GHz	Itanium 800MHz
Main	$7.0\mu s$ $(E_p \geq 16K)$	$6.6\mu s$ $(E_p \geq 32K)$	$22.2\mu s$ $(E_p \geq 8K)$
Main/Cache	$0.98*Ln(E_p) - 3.4\mu s$ $(800 < E_p < 16K)$	$0.76*Ln(E_p) - 2.4\mu s$ $(1600 < E_p < 32K)$	$3.7*Ln(E_p) - 9.3\mu s$ $(400 < E_p < 8K)$
Cache	$3.0\mu s$ $(E_p \leq 800)$	$2.4\mu s$ $(E_p \leq 1600)$	$11.4\mu s$ $(E_p \leq 400)$

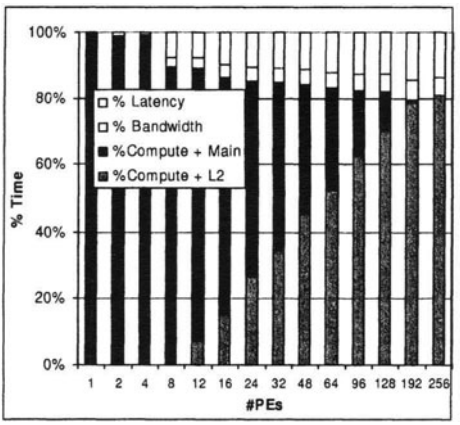


Figure 7. Time Component predictions improvement (Reac mesh on HP ES40).

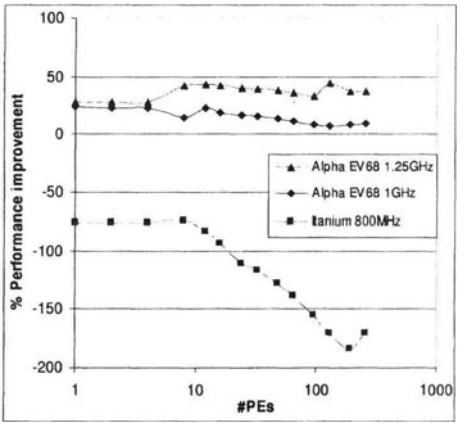


Figure 8. Predicted performance using Alpha EV68 1Ghz and Itanium 800MHz.

The time component predictions (Figure 7) are illustrated for the Reac mesh with $MCPS$ set at 64. A lower value of $MCPS$ results in more communications. However, it can be seen in Figure 7 that the application remains compute bound, with latency dominating the bandwidth component of the communication. In addition, due to the strong scaling behavior, the subgrid sizes get smaller at higher processor counts leading to better L2 cache behavior. For higher values of $MCPS$, less communication occurs and hence the overall time comprises a smaller communication component.

The predicted performance improvement over an Alpha EV68 833MHz system when using a system with either Alpha EV68 1GHz or an Itanium 800MHz reveals some interesting features (Figure 8). The 1GHz Alpha outperforms the

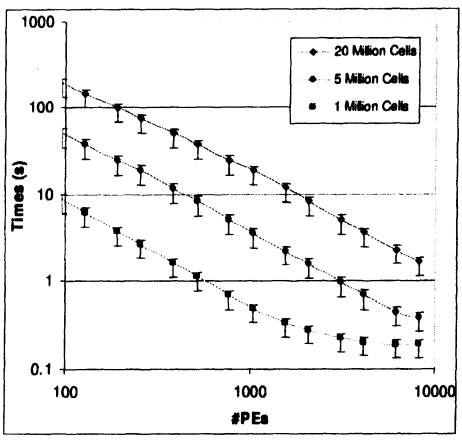


Figure 9. Predicted performance on larger meshes and processor counts.

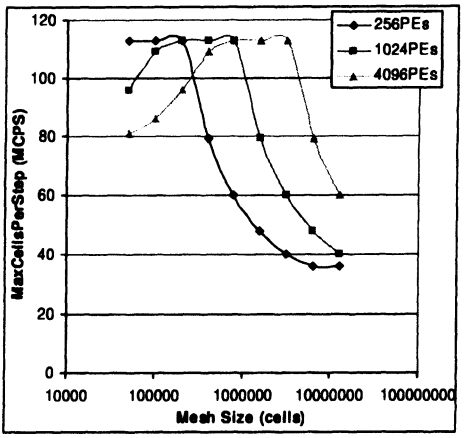


Figure 10. Predicted value of $MCPS$ (communication cost < 20% of total).

833Mhz processor as expected by about 20% for larger subgrid sizes (not fitting in the cache), but decreases to about 10% on larger PE counts due to similar L2 cache performance for the two processors. Similarly the 1.25GHz Alpha also outperforms the 833MHz EV68 due in part to its increased L2 cache (16MB vs. 8MB). The Itanium, at similar clock speeds, is performing significantly poorer than the Alpha. Its cache performance is also much poorer than that of the Alpha's, as evident by the lower performance on larger processor counts.

The predicted performance for larger meshes on larger processor counts indicates an expected good scaling behavior (Figure 9). This study considers meshes of size: 1,000,000 cells, 5,000,000 cells, and 20,000,000 cells being processed on up to 8192 PEs. Each curve assumes a PCE of 0.8 with the min and max bars indicating a PCE of 0.6 and 0.9 respectively. It can be seen that the smaller mesh has a significant communication component at large PE counts, as shown by the knee in the curve. If a value of $MCPS$ greater than 512 was used, the number of steps would decrease and hence the amount of communication would decrease. This indicates that for larger meshes, a corresponding larger value of $MCPS$ should be utilized.

The usage of the GM model to determine a value of $MCPS$ so that the communication component of the runtime is at most 20% is shown in Figure 10. A lower value of $MCPS$ is beneficial as it will result in a higher PCE value but increased communication cost. It can be seen that the value of $MCPS$ is dependent upon the mesh size as well as the processor count. It can be seen that the value of $MCPS$ actually exhibits an increase before decreasing as the mesh increases with size for a particular processor count. As the number of processors increases, the curve can be seen to shift to the right. The form of

these curves results from two competing factor. The first factor is the increased computation time as the mesh size increases due to less cache reuse - this leads to an increase in communication time resulting in a smaller $MCPS$ value. The second factor is the pipeline length as the mesh gets smaller - on a small mesh the pipeline length can dominate the PCE. Thus a greater degree of blocking (a lower value of $MCPS$) is required to keep processor utilization high.

From these analyses it can be seen that given the complexity of the performance issues associated with Tycho, this performance space cannot be analyzed without a general model. Issues such as determining the value of $MCPS$ are often too complicated and result from an interplay of many factors. It should be noted that it is planned to incorporate a version of the GM into the application code in order to dynamically determine the value of $MCPS$ at runtime and thus help optimize the time- to-solution for Tycho.

7. Summary

In this work we have presented predictive performance and scalability models for a deterministic transport application using unstructured meshes. The models take into account the main computation and communication characteristics of the entire code. Two of the models developed are analytic whereas a third is based on the analysis of runtime traces. The models are shown to have good accuracy through validation on a 64 node HP AlphaServer system.

The models varied in their degree of generality. The validation showed that as the models incorporated more specific details on the actual mesh being processed, the accuracy increased. However this also results in loss of generality, limiting the amount of insight into achievable performance.

It was shown that a generally applicable analytical model of this application can encapsulate the performance characteristics with reasonably accuracy and then used to explore the expected performance in many different performance scenarios. This is a key element of developing application based performance models - that is to explore the performance space - to aid in the development of the application code and to predict performance on future system architecture prior to their availability for measurement.

We believe performance modeling is key to building performance engineered applications and architectures. This work is one of few performance models that exist for entire applications. It follows on from our work on structured particle transport modeling [3], adaptive mesh refinement modeling [6], and Monte-Carlo simulation [9].

Acknowledgments

Los Alamos National Laboratory is operated by the University of California for the National Nuclear Security Administration of the US Department of Energy. The work was funded by LDRD at the Los Alamos National Laboratory.

References

[1] W.D. Gropp, D.K. Kaushik, D.E. Keyes and B.F.Smith. Performance Modeling and Tuning of an Unstructures Mesh CFD Application. In *Proc. SC2000*, Dallas, 2000.

[2] S. Girona, J. Labarta and R.M. Badia. Validation of Dimemas communication model for MPI collective operations. In *Proc. EuroPVM/MPI'2000, LNCS*, 1908:39-46, Springer-Verlag, 2000.

[3] A. Hoisie, O. Lubeck and H. Wasserman. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *Int. J. of High Performance Computing Applications*, 14:330-346, 2000.

[4] A. Hoisie, O. Lubeck, H.J. Wasserman, F. Petrini, H.J. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proc. ICPP 2000*, August 20-25, 2000. Toronto, Canada.

[5] G. Karypis, V. Kumar. *METIS 4.0: Unstructured Graph Partitioning and Sparse Matrix Ordering System.* Technical Report, Department of Computer Science, University of Minnesota, 1998.

[6] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H.J. Wasserman, M.L. Gittings. Predictive Performance and Scalability Modeling of a Large-scale Application. In *Proc SC2001*, Denver, November 2001.

[7] D.J. Kerbyson, E. Papaefstathiou, J.S. Harper, S.C. Perry, G.R. Nudd. Is Predictive Tracing Too late for HPC Users? In R.J. Allan, A. Simpson, and D.A. Nicole (Eds), *High Performance Computing*, Plenum Press, pages 57-67, March 1999.

[8] K.R. Koch, R.S. Baker, R.E. Alcouffe. *A Parallel Algorithm for 3D Sn Transport Sweeps.* LA-CP-92-406, Los Alamos National Laboratory, 1992.

[9] M.M. Mathis, D.J. Kerbyson, A. Hoisie. A Performance Model of Non-deterministic Particle Transport on Large-Scale Systems, In *Proc. Computational Science - ICCS 2003, LNCS*, 2659:905-915, Springer-Verlag, 2003.

[10] G.R. Nudd, D.J. Kerbyson et.al. PACE: A Toolset for the Performance Prediction of Parallel and Distributed Systems. *Int. J. of High Performance Computing Applications*, 14:228-251, 2000.

[11] E. Papaefstathiou, D.J. Kerbyson. Predicting Communication Delays of Detailed Application Workloads. In *Proc of 13th ISCA Int. Conf. on Parallel and Distributed Computing Systems (PDCS)*, Las Vegas, August 2000.

[12] S.D. Pautz. An Algorithm for Parallel Sn Sweeps on Unstructures Meshes. *J. Nuclear Science and Engineering*, Vol. 140, pages 111-136, 2002.

[13] F. Petrini, W.C. Feng, A. Hoisie, S. Coll, E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46-57, 2002

[14] S. Plimpton, B. Hendrickson, S. Burns, W. McLendon. Parallel Algorithms for Radiation Transport on Unstructured Grids. In *Proc. SC2000*, Dallas, November 2000.

PERFORMANCE OPTIMIZATION OF RK METHODS USING BLOCK-BASED PIPELINING

Matthias Korch
Department of Mathematics and Physics
University of Bayreuth, Bayreuth, Germany
matthias.korch@uni-bayreuth.de

Thomas Rauber
Department of Mathematics and Physics
University of Bayreuth, Bayreuth, Germany
rauber@uni-bayreuth.de

Gudula Rünger
Department of Computer Science
Technical University of Chemnitz, Chemnitz, Germany
ruenger@informatik.tu-chemnitz.de

Abstract The efficiency of modern microprocessors is extremely sensitive towards the structure and memory access pattern of programs to be executed. This is caused by memory hierarchies which were introduced to reduce average memory access times. In this paper, we consider embedded Runge-Kutta (RK) methods for the solution of ordinary differential equations arising from space discretization problems for partial differential equations and study their efficient implementation on modern microprocessors. Different program variants with different execution orders and storage schemes are investigated. In particular, we explore how the potential parallelism in the stage vector computation can be exploited in a pipelining approach in order to improve the locality behavior of the RK implementations. Experiments show that this results in efficiency improvements on several recent processors.

Keywords: performance optimization, ordinary differential equations, Runge-Kutta methods, locality improvement, pipelining

1. Introduction

Time-dependent partial differential equations (PDEs) with initial conditions can be solved by discretizing the spatial domain using the method of lines. This leads to an initial value problem (IVP) for a system of first order ordinary differential equations (ODEs) in the time domain of the form

$$\mathbf{y}'(x) = \mathbf{f}(x, \mathbf{y}(x)) \text{ with } \mathbf{y}(x_0) = \mathbf{y}_0 \tag{1}$$

where $\mathbf{y}_0$ is the initial vector at start time x_0, $n \geq 1$ is the system size, and $\mathbf{f} : \mathbf{R} \times \mathbf{R}^n \rightarrow \mathbf{R}^n$ is the right hand side function describing the structure of the ODE system. For the numerical integration of problem (1) approximating the solution $\mathbf{y} : \mathbf{R} \rightarrow \mathbf{R}^n$, an error control mechanism can be used to adapt the stepsize so that a given accuracy can be guaranteed. Thus the number of discretization points in the time domain is usually quite small compared to a global discretization that also discretizes the time domain with a fixed stepsize (resulting in a linear or nonlinear algebraic equation system).

RK methods with embedded solutions are one of the most popular one-step methods for the numerical integration of non-stiff IVPs because they combine low execution times with good numerical properties. An embedded RK method computes two approximations of different convergence order with the same evaluations of the right hand side function of the ODE system and uses them for stepsize control [10]. Examples are the methods of Fehlberg [7] and Dormand&Prince [11]. Variants of these methods are used in many software packages like the subroutine DVERK in IMSL or RKSUITE [4].

Modern microprocessors exhibit a complex architecture with multiple functional units, to which instructions are dynamically dispatched, and a storage hierarchy with registers, two or three caches of different size and associativity, and a main memory. Memory hierarchies provide improved average memory access times due to the locality of reference principle. As a consequence, spatial locality and temporal locality have a large influence on the execution time of a program.

In this article, we use embedded RK methods to solve ODE systems that arise from PDEs by the method of lines. Those ODE systems have a specific structure with a low coupling density induced by the coupling of the original PDE system. We address the question how the operations of the RK method within one time step should be arranged such that temporal and spatial locality of the resulting memory access pattern are high.

The starting point are investigations for general purpose RK solvers for non-stiff ODEs with step-size control where no assumptions about the access structure of the right hand side functions $\mathbf{f}$ are taken into account. This is the situation of black-box solvers provided by scientific libraries. Efficient implementations can be obtained by exploiting parallelism [12] or by tuning towards an efficient exploitation of the memory hierarchy [13]. In this paper, program

transformations for locality improvements are resumed and further transformations and storage schemes are applied. In particular, we consider how specific access structures of $\mathbf{f}$ imposed by grid-based problems can be exploited to further increase the locality of memory references, and we use the specific access structure of $\mathbf{f}$ for a pipelined computation of approximation vectors. This is the basis for a reordering of the computation such that the working space of the algorithm is significantly decreased, which leads to a better locality behavior.

The contribution of this paper is to develop reorderings and transformations of the computation structure of embedded RK methods that lead to a better exploitation of the processor architecture. All reorderings and transformations preserve the dependence structure of the computations. There are no assumptions about the specific characteristics of the cache memory like cache size, cache line size, cache associativity, or cache replacement strategy. We show that the new computation schemes lead to significant reductions of the execution time on modern microprocessors like the Pentium III or the UltraSPARC III.

The rest of the paper resumes different locality sensitive code realizations for RK methods in Section 2. Section 3 considers computation structures resulting from grid-based computations and presents a pipelining approach for the realization. Section 4 shows the resulting execution times on different architectures. Section 5 discusses related work and Section 6 concludes.

2. Computational Structure

For non-stiff ODE systems of the form (1), explicit RK methods with an error control and stepsize selection mechanism are robust and efficient and guarantee that the obtained approximation of $\mathbf{y}$ is consistent with a predefined error tolerance [10, 6]. In each time step, these methods compute a discrete approximation vector $\eta_{\kappa+1} \in \mathbf{R}^n$ for the solution function $\mathbf{y}(x_{\kappa+1})$ at position $x_{\kappa+1}$ using the previous approximation vector η_κ. We consider an s-stage RK method that computes s stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s \in \mathbf{R}^n$ according to

$$\mathbf{v}_l = \mathbf{f}(x_\kappa + c_l h_\kappa, \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li}\mathbf{v}_i), \qquad l = 1, \ldots, s, \tag{2}$$

and uses the stage vectors to compute the approximation vectors

$$\eta_{\kappa+1} = \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l \mathbf{v}_l, \quad \text{and} \quad \hat{\eta}_{\kappa+1} = \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} \hat{b}_l \mathbf{v}_l \tag{3}$$

where $\hat{\eta}_{\kappa+1}$ is an additional vector for error control. The s–dimensional vectors $b = (b_1, \ldots, b_s)$, $\hat{b} = (\hat{b}_1, \ldots, \hat{b}_s)$, $c = (c_1, \ldots, c_s)$ and the $s \times s$ matrix $A = (a_{li})$ specify the particular RK method. The order r of approximation $\eta_{\kappa+1}$ and the order $\hat{r}$ of approximation $\hat{\eta}_{\kappa+1}$ usually differ by 1, e.g., $r = \hat{r} + 1$. The

difference between the two approximations $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ gives an asymptotic estimate of the local error in the lower order approximation and is used for stepsize control [6].

Given an embedded RK method with appropriate numerical properties, we consider different execution orders of the same code and investigate the impact on the resulting execution time for processors with memory hierarchy. Improvements of the execution time can be achieved by global rearrangement of data accesses. For those rearrangements, all data dependencies in (2) and (3) between the vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$, and $\eta_{\kappa+1}$ have to be taken into account in order to preserve correctness. For an arbitrary right hand side function $\mathbf{f}$, we are required to make the conservative assumption that every component of $\mathbf{f}$ depends on all vector components of its argument vector, i.e., the computation of one component of $\mathbf{v}_i$ requires that all components of $\mathbf{v}_1, \ldots, \mathbf{v}_{i-1}$ are already determined. For this case, several implementation versions of general embedded RK methods have been derived [13], see Figure 1 for an illustration of the iteration space:

(A) In a straightforward implementation of (2) and (3), the body of the iteration over time contains a non-tightly nested loop structure consisting of a loop over the stage vector computation with the computation of the argument vector $\mathbf{w}$ as inner loop and the loop over vector components of the stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$ as innermost loop.

(B) (Figure 2) To prepare later loop restructurings, separate argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ are introduced as argument vectors for different stage vector computations in order to decouple the dependencies. In the program code, the stages are computed successively. At each stage $i = 1, \ldots, s$ a nested loop is executed that computes the elements of the argument vectors $\mathbf{w}_i$ by calculating the weighted sum of the stage vectors $\mathbf{v}_j$, $j = 1, \ldots, i-1$. The working space of stage i of this implementation consists of $i \cdot n$ stage vector elements, n argument vector elements, and n elements of the approximation vector.

(C) To use stage vector components as soon after their computation as possible, the loops computing argument vectors and stage vectors are interchanged. Now a stage vector $\mathbf{v}_l$ is first computed and then immediately used to build all argument vectors for succeeding function evaluations in the same time step. All interleaved accesses to vectors $\mathbf{v}_l$ are removed and actually only one vector $\mathbf{v}$ is needed to perform the computation of the different stage vectors one after another.

(D) Further optimizations are possible by a loop interchange with the dimension loop. Now only one scalar variable is used to represent all stage vector components. The transformation does not only lead to a better temporal

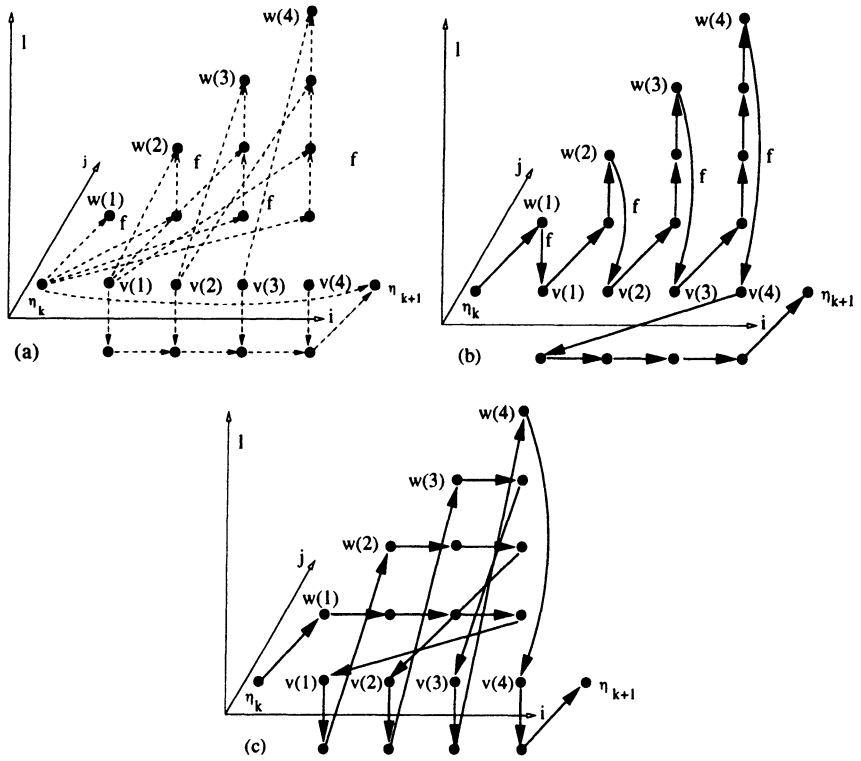


Figure 1. Illustration of data dependencies (a) and execution orders (b) and (c) in implementation version (B) and (C) for $s = 4$. Nodes represent intermediate vector results of computations (the system dimension $j = 1, \ldots, n$ is not shown explicitly). The nodes in the bottom row show the accumulation of the approximation vector $\eta_{\kappa+1}$, the nodes in row above show the original stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$. The top nodes of the columns denote the modified stage vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$. The dashed arrows in (a) show data dependencies, the solid arrows in (b) and (c) show the order of the computations.

locality but also reduces the number of storage locations used within the program which may lead to further reduction of the access distances.

(E) By considering a specific RK method with a fixed number of stages, further optimizations like the unrolling of loops over the stages and direct access to the RK coefficients can be performed, leading to a specialized efficient realization, e.g., for the DOPRI5 method.

The descriptions of the program versions (A)–(E) outline the main steps of the program transformations only. There are more intermediate restructuring and transformation steps needed for the entire transformation process.

The program versions (C) and (D) correspond to a computation scheme with delayed function evaluations in computation scheme (2) and (3) in which a

```
for (i = 0; i < s; i++) {
    for (j = 0; j < n; j++) w_i[j] = η[j];
    for (l = 0; l < i; l++)
        for (j = 0; j < n; j++) w_i[j] += h a_il v_l[j];
    for (j = 0; j < n; j++) v_i[j] = f_j(x + c_i h, w_i); }
for (j = 0; j < n; j++) { t[j] = 0.0; u[j] = 0.0; }
for (i = 0; i < s; i++)
    for (j = 0; j < n; j++) { t[j] += b̃_i v_i[j]; u[j] += b_i v_i[j]; }
for (j = 0; j < n; j++) { η[j] += h u[j]; ε[j] = h t[j]; }
```

Figure 2. Implementation (B). The vector $\tilde{\mathbf{b}}$ is defined by $\tilde{\mathbf{b}} = \mathbf{b} - \hat{\mathbf{b}}$.

function evaluation is started not before its result is needed for another computation. This computation scheme results from applying the transformation $\mathbf{v}_i = \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i)$, $i = 1, \ldots, s$, to (2) and (3) and results in:

$$\mathbf{w}_l = \eta_\kappa + h_\kappa \sum_{i=1}^{l-1} a_{li} \mathbf{f}(x_\kappa + c_i h_\kappa, \mathbf{w}_i), \quad l = 1, \ldots, s. \tag{4}$$

The approximation vectors $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ are then computed as:

$$\begin{aligned} \eta_{\kappa+1} &= \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} b_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l), \quad \text{and} \\ \hat{\eta}_{\kappa+1} &= \eta_\kappa + h_\kappa \cdot \sum_{l=1}^{s} \hat{b}_l \mathbf{f}(x_\kappa + c_l h_\kappa, \mathbf{w}_l). \end{aligned} \tag{5}$$

After the evaluation of one component of $\mathbf{f}(\mathbf{w}_i)$, the computation scheme (4) also allows the update of the corresponding components of all vectors $\mathbf{w}_j$, $j > i$, and of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. Therefore, no explicit storage of the results of the function evaluations is necessary and temporal locality for the function value and spatial locality for the updated vectors is established. Implementation (B) can be considered to be using both computation schemes (2)/(3) and (4)/(5) since it stores all stage vectors as well as all argument vectors.

3. Exploiting Specific Access Structures

The assumption about a general $\mathbf{f}$ restricts program restructuring. But if $\mathbf{f}$ is given, the specific access structure of $\mathbf{f}$ can be used for further reordering of the computations. Many application problems are described by a right hand side function $\mathbf{f} = (f_1, \ldots, f_n)$ with component functions f_l which actually need only a few components of the entire argument vector depending on its index $l = 1, \ldots, n$. We consider a reaction-diffusion equation as a typical example for an ODE system that arises from applying the method of lines to a

time-dependent PDE. The 2D-Brusselator equation

$$\frac{\partial u}{\partial t} = 1 + u^2 v - 4.4u + \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$
$$\frac{\partial v}{\partial t} = 3.4u - u^2 v + \alpha \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} \right)$$

for $0 \leq x \leq 1$, $0 \leq y \leq 1$, $t \geq 0$, describes the reaction of two chemical substances with a diffusion term [10]. The unknown functions u and v describe the concentrations of the two substances. A Neumann boundary condition $\frac{\partial u}{\partial \eta} = 0, \frac{\partial v}{\partial \eta} = 0$, and the initial conditions $u(x, y, 0) = 0.5 + y$, $v(x, y, 0) = 1 + 5x$ are used. A standard five-point-star discretization of the spatial derivatives on a uniform $N \times N$ grid with mesh size $1/(N-1)$ leads to an ODE system of dimension $2N^2$ for the discretized solution $\{U_{ij}\}_{i,j=1,\ldots,N}$ and $\{V_{ij}\}_{i,j=1,\ldots,N}$, which is a non-stiff ODE system for $\alpha = 2 \cdot 10^{-3}$.

Storage schemes. We consider two different linearizations of the grid points $\{U_{ij}\}_{i,j=1,\ldots,N}$ and $\{V_{ij}\}_{i,j=1,\ldots,N}$. The *row-oriented organization*

$$U_{11}, U_{12}, \ldots U_{NN}, V_{11}, V_{12}, \ldots, V_{NN} \tag{6}$$

results in function component f_l accessing argument components $l - N, l - 1, l, l + 1, l + N$, and $l + N^2$, if available, for $l = 1, \ldots, N^2$ or components $l - N, l - 1, l, l + 1, l + N$, and $l - N^2$, if available, for $l = N^2 + 1, \ldots, 2N^2$. This is a typical access structure for grid-based computations. A specific disadvantage concerning the locality of memory references is that for the computation of each component f_l a component of the argument vector in distance N^2 is accessed due to the coupling in (3). A *mixed row-oriented organization*

$$U_{11}, V_{11}, U_{12}, V_{12}, \ldots, U_{ij}, V_{ij}, \ldots U_{NN}, V_{NN}. \tag{7}$$

stores corresponding components of U and V next to each other and results in function component f_l accessing argument components $l - 2N, l - 2, l, l + 1, l + 2, l + 2N$ (if available) for $l = 1, 3, \ldots, 2N^2 - 1$ and $l - 2N, l - 2, l - 1, l, l + 2, l + 2N$ (if available) for $l = 2, 4, \ldots, 2N^2$. For this access structure the most distant components of the argument vector to be accessed for the computation of one component of $\mathbf{f}$ have distance $2N$.

Pipelining. For computation scheme (4)/(5) with storage scheme (7), a pipelined computation based on a division of the stage, the argument and the approximation vectors into N blocks of size $2N$ each can be exploited. The computation of an arbitrary block $J \in \{1, \ldots, N\}$ of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ requires the corresponding block J of $\mathbf{v}_s$, which itself depends on block J of $\mathbf{w}_s$ and, if available, the neighboring blocks $J - 1$ and $J + 1$ of $\mathbf{w}_s$ because of the access

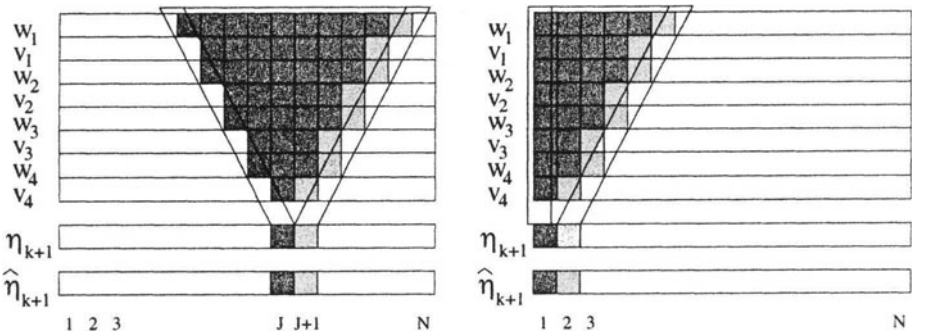


Figure 3. *Left:* Dependence structure for storage scheme (7) in the case $s = 4$. If block J of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ has been computed previously, the computation of block $J+1$ requires accessing one additional block of each of the stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_4$ and the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_4$ only. *Right:* Blocks accessed to compute the first and the second blocks of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$.

pattern of $\mathbf{f}$. The computation of the blocks $J-1$, J, $J+1$ of $\mathbf{w}_s$ requires the corresponding blocks of $\mathbf{v}_{s-1}$. But these blocks cannot be computed before the computation of the blocks $J-2$ to $J+2$ of $\mathbf{w}_{s-1}$ is finished. Altogether, each block J of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ depends on at most $\sum_{i=1}^{s}(2i+1) = s(s+1)+s = s(s+2)$ blocks of $\mathbf{w}_1, \ldots, \mathbf{w}_s$ of size $2N$ and $\sum_{i=1}^{s}(2i-1) = s(s+1)-s = s^2$ blocks of $\mathbf{v}_1, \ldots, \mathbf{v}_s$ of size $2N$, see Figure 3 (left).

This dependence structure can be exploited in a pipelined computation order for the blocks of the stage vectors $\mathbf{v}_1, \ldots, \mathbf{v}_s$ and the argument vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ in the following way: the computation is started by computing the first $s+1$ blocks of argument vector $\mathbf{w}_1$. Since the computation of component $(\mathbf{v}_1)_l$ requires the evaluation of $f_l(x_\kappa, \mathbf{w}_1)$ and since $\mathbf{f}$ has the specific access structure described above, the computation of s blocks of $\mathbf{v}_1$ is enabled, which again enables the computation of s blocks of $\mathbf{w}_2$ and so on. Finally, one block of $\mathbf{v}_s$ is computed and used to compute the first block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. The next block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ can be determined by computing only one additional block of $\mathbf{w}_1$ which enables the computation of one additional block of $\mathbf{v}_1, \ldots, \mathbf{v}_s$ and $\mathbf{w}_2, \ldots, \mathbf{w}_s$, see Figure 3 (right). This computation is repeated until the last blocks of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ are computed. Figure 4 (left) shows the iteration space of the pipelined computation scheme. The boxes attached to nodes illustrate the vector dimension of stage vectors and approximation vectors.

Working Space. The advantage of the pipelining approach is that only those blocks of the argument vectors are kept in the cache which are needed for further computations of the current step. One step of the pipelining computation scheme computes s stage vector blocks, s argument blocks and one block of

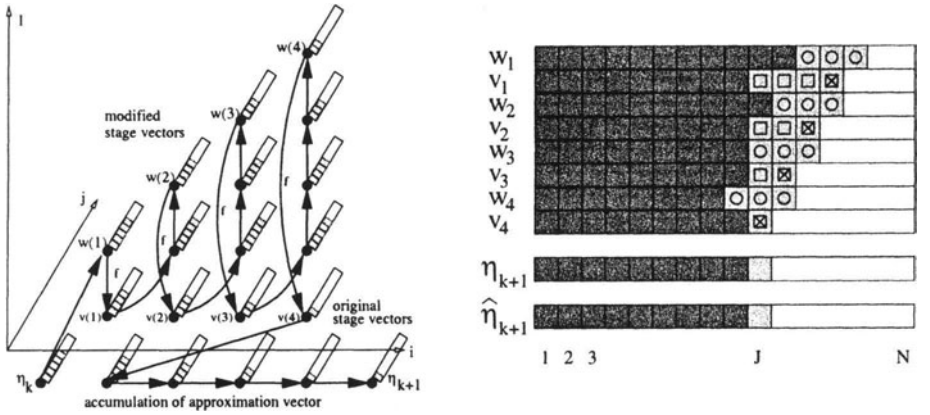


Figure 4. *Left:* Illustration of pipelined computation for $s = 4$. The dimension of the vectors is shown to demonstrate the pipelined computation from Figure 3. Filled boxes denote blocks of (intermediate) result vectors. The first block of $\eta_{\kappa+1}$ depends on all filled blocks shown in the figure. The filling structure of the vectors $\mathbf{w}_1, \ldots, \mathbf{w}_s$ shows the triangular structure given in Figure 3 (right). *Right:* Illustration of the working space of one pipelining step. Argument blocks marked by a circle are accessed during the function evaluation executed to compute the stage vector blocks tagged by a cross. Stage vector blocks used to compute blocks of argument and approximation vectors are marked by a square.

$\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. Since the computation of one block J of one stage vector accesses the blocks $J-1$, J, and $J+1$ of the corresponding argument vector, altogether $3s$ argument blocks must be accessed to compute one block of $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$. Additionally, $\sum_{i=1}^{s} i = s(s+1)/2$ blocks of the stage vectors are accessed because the computation of one argument block J requires the blocks J of all previous stage vectors. Consequently, the working space of the pipelining computation scheme consists of $2 + 3s + s(s+1)/2$ blocks of size $2N$, see Figure 4 (right). For the DOPRI5 method with $s = 7$ stages, at most 51 blocks would have to be kept in cache to minimize the number of cache misses. This is usually a small part of the N blocks of size $2N$ that each stage vector contains. Taking $\eta_{\kappa+1}$ and $\hat{\eta}_{\kappa+1}$ into consideration, the proportion of the total number of blocks that have to be held in cache is

$$\frac{(2 + 3s + s(s+1)/2)}{(2s+2)N} = O\left(\frac{s}{4N}\right)$$

with usually $s \ll N$.

Implementation. The pipelining approach has been implemented in the following two program variants:

(F) The main body (Fig. 5 (a)) of the implementation consists of three phases: initialization of the pipeline, diagonal sweep over the argument vectors,

```
k = s · 2N;

for (j = 0; j < k; j += 2N)
    STAGE0(j);

for (i = 1, l = k − 2N; i < s; i++, l -= 2N)
    for (j = 0; j < l; j += 2N)
        STAGE(j, i);

for (j = k, l = k + 2N; j < n, j += l)
{
    STAGE0(j);
    for (i = 1, j -= 2N; i < s; i++, j -= 2N)
        STAGE(j, i);
    FINAL(j);
}

for (l = 1, k = n − 2N; l < s; l++)
{
    for (i = l, j = k; i < s; i++, j -= 2N)
        STAGE(j, i);
    FINAL(j);
}
FINAL(k);
```

(a) Body

```
STAGE0(A):
for (p = A; p < A + 2N; p++)
    w0[p] = η[p];

STAGE(A, m):
for (p = A; p < A + 2N; p++)
    wm[p] = η[p];
for (p = A; p < A + 2N; p++)
    vm−1[p] = fp(x + cm−1h, wm−1);
for (r = 0; r < m; r++)
    for (p = A; p < A + 2N; p++)
        wm[p] += h·amr·vr[p];

FINAL(A):
for (p = A; p < A + 2N; p++) {
    vs−1[p] = fp(x + cs−1h, ws−1);
    t[p] = 0.0; u[p] = 0.0; }
for (r = 0; r < s; r++)
    for (p = A; p < A + 2N; p++) {
        t[p] += br·vr[p];
        u[p] += br·vr[p]; }
for (p = A; p < A + 2N; p++) {
    η[p] += h·u[p];
    ε[p] = h·t[p]; }
```

(b) Macros

Figure 5. Implementation (F). The vector $\tilde{\mathbf{b}}$ is defined by $\tilde{\mathbf{b}} = \mathbf{b} - \hat{\mathbf{b}}$.

and finalization of the pipeline. We introduce the following three macros (Fig. 5 (b)): STAGE0(A) is used to compute one block of vector $\mathbf{w}_0$. Starting at offset A, STAGE(A, m) computes one block of the stage vector $\mathbf{v}_{m-1}$ and one block of the argument vector $\mathbf{w}_m$. The macro FINAL(A) evaluates the function values of one block of the last argument vector to obtain the corresponding stage vector block and finally computes one block of $\eta_{\kappa+1}$ and one block of the local error estimate $\epsilon_{\kappa+1} = \eta_{\kappa+1} - \hat{\eta}_{\kappa+1}$.

(G) The second implementation is a pipelined version of the specialized implementation, which is optimized for a fixed number of $s = 7$ stages and exploits locality in the solution of Brusselator-like systems.

4. Runtime experiments

In this section, we investigate the performance enhancements achieved by the pipelining approach described in the previous section and compare the results with the general implementations for the two different storage schemes of the Brusselator function. For these investigations, different target platforms with varying memory hierarchies have been used. In particular, we consider the following systems:

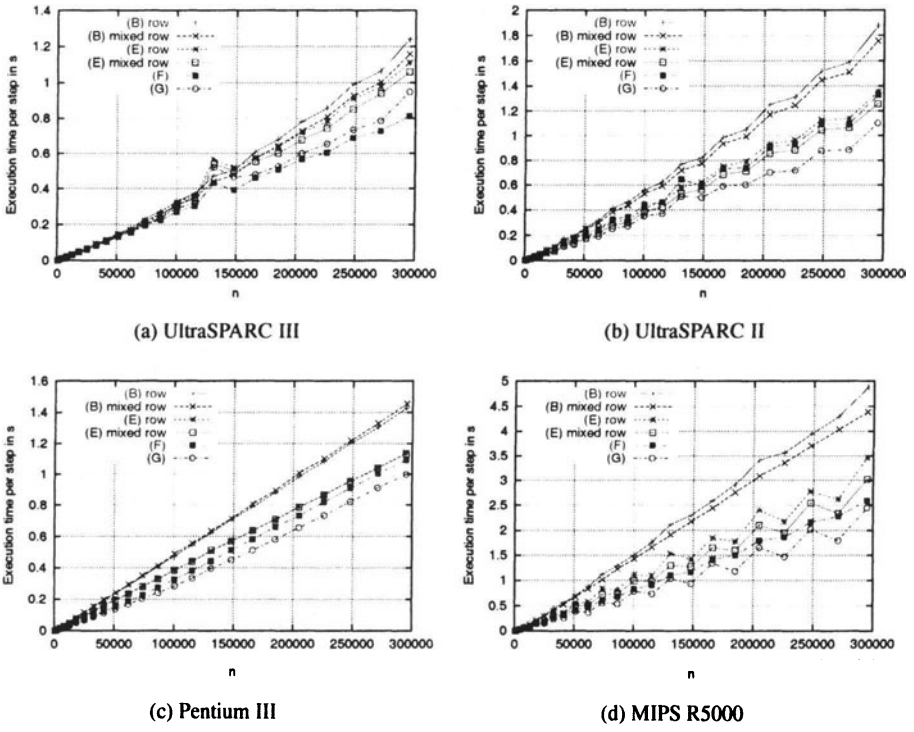


Figure 6. Execution times of the RK implementations.

1. UltraSPARC III at 750 MHz, 64 KB L1 data cache (4-way associative), 32 KB L1 instruction cache (4-way ass.), 8 MB L2 cache (2-way ass.),

2. UltraSPARC II at 450 MHz, 16 KB L1 data cache (1-way ass.), 16 KB L1 instruction cache (2-way ass.), 4 MB L2 cache (1-way ass.),

3. Pentium III at 600 MHz, 16 KB L1 data cache (4-way ass.), 16 KB L1 instruction cache (4-way ass.), 256 KB L2 cache (8-way ass.),

4. MIPS R5000 at 300 MHz, 32 KB L1 data cache (2-way ass.), 32 KB L1 instruction cache (2-way ass.), 1 MB L2 cache.

All programs have been implemented in C and use double precision. As RK method, we use the DOPRI5 method with $s = 7$ stages. Figure 6 shows the execution times per step for the Brusselator system on the target systems introduced above. In Figures 7 and 8 we show the number of instructions executed and the L1 and L2 cache misses measured on the UltraSPARC III and the Pentium III system with help of the PCL library [2].

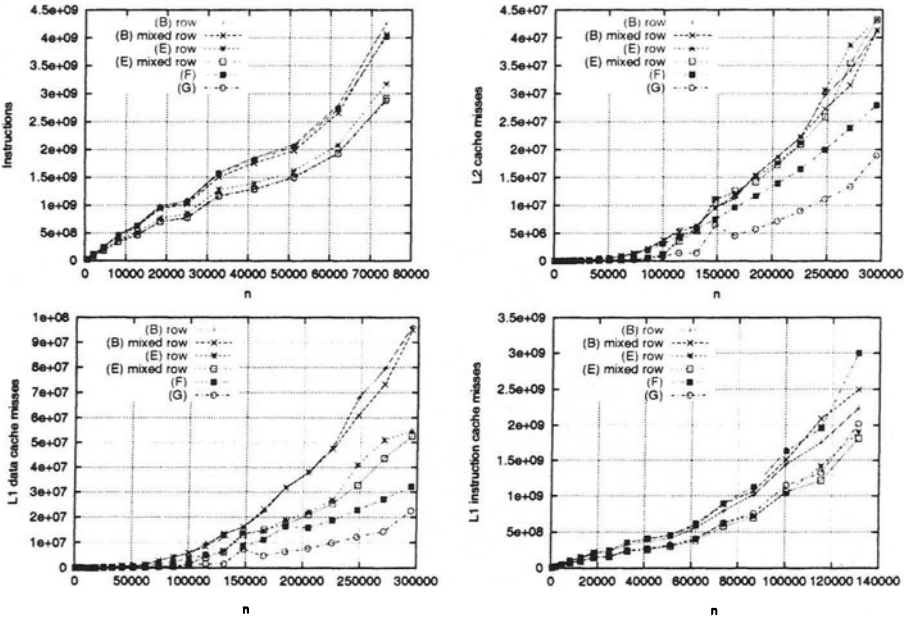


Figure 7. Cache behavior and instructions executed on UltraSPARC III.

Comparison of storage schemes. The general implementations (A)–(E) can be used to solve arbitrary ODE systems. Since the mixed row-oriented storage scheme (7) of the Brusselator system reduces the maximum distance of the components accessed for the computation of one component of $\mathbf{f}$, it can be hoped that this increased spatial locality accelerates the execution of the general implementations. We have compared the execution times of the two storage schemes (6) and (7) using the general implementations (A)–(E). The results for the implementations (B) and (E) are included in Figures 6, 7 and 8.

On all machines, except for the Pentium III system, the mixed row-oriented storage scheme is significantly faster than the pure row-oriented scheme. The best results have been obtained on the MIPS processor. On this processor the use of the mixed row-oriented storage scheme leads to 10.13 % faster execution times for implementation (B) and 12.77 % for implementation (E) when the size of the system is $n = 294912$. The execution times on the Pentium III system are very similar to each other for both storage schemes.

Figure 7 shows that the numbers of cache misses on the UltraSPARC III are very similar for both storage schemes. There are only slight improvements for the L2 and L1 data cache misses. The number of L1 instruction cache misses for implementation (B) with the mixed row-oriented storage scheme is even noticeably higher than the number measured with the original storage scheme. Thus, the improvements of the execution times achieved with the mixed row-

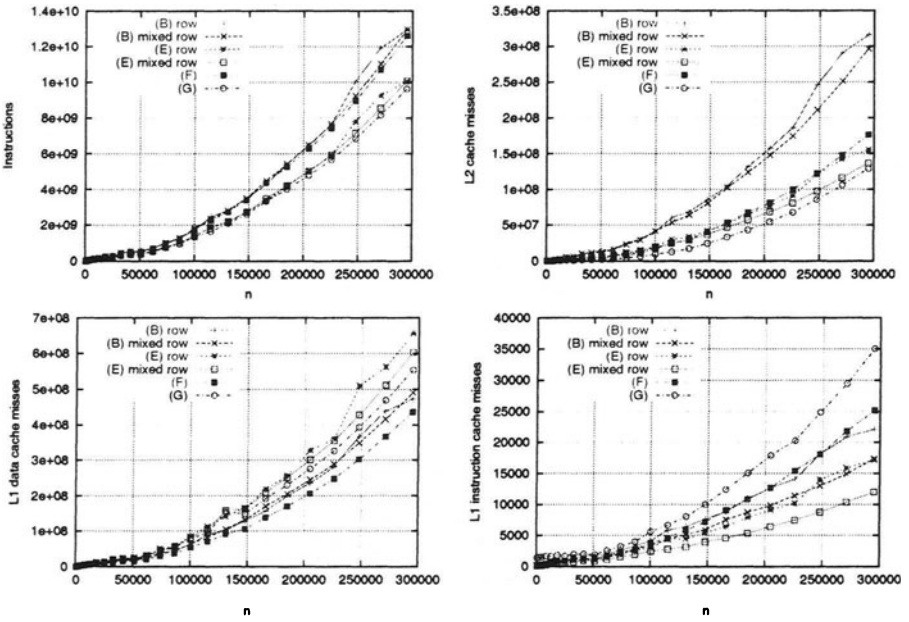


Figure 8. Cache behavior and instructions executed on Pentium III.

oriented storage scheme on this machine seem to be caused by the lower number of instructions executed. The difference in the numbers of instructions executed for both storage schemes is caused by the different codes of the function **f** and, as a consequence, the different numbers of machine instructions the two implementations of **f** are compiled to.

On the Pentium III system the numbers of instructions executed for both storage schemes do not differ significantly. But the numbers of misses in the L2 cache and the L1 instruction cache are remarkably reduced for the mixed row-oriented storage scheme. The differences between the L1 data cache misses are smaller than those of the other caches. Implementation (B) has even fewer L1 data cache misses with the pure row-oriented storage scheme for most system sizes. Further experiments to compare the execution times for the two storage schemes have been performed with the implementations (A), (C) and (D). They confirm the results measured for the implementations (B) and (E).

Pipelining. The pipelining approach reduces the execution times on all machines we have considered. Again the best results have been obtained on the MIPS processor. On this machine, implementation (F), which is specialized for the mixed row-oriented ordering, outperformed the general implementation (B) by 41.00 % for system size $n = 294912$. Implementation (G) ran 18.94 % faster than the corresponding general implementation (E). On the other machines im-

plementation (F) still was 23 % to 29 % faster than implementation (B), and implementation (G) was 10 % to 12 % faster than implementation (E). As expected, the enhanced locality of the pipelining approach leads to reduced L2 cache misses on the UltraSPARC III as well as the Pentium III processor. On the UltraSPARC III system the number of L1 data cache misses is also decreased. The number of L1 instruction cache misses does not change significantly on the UltraSPARC III, but is increased on the Pentium III machine. Similarly, the number of instructions executed is hardly unchanged on the UltraSPARC III, but is slightly smaller on the Pentium.

5. Related Work

Because of their large impact on the performance, optimizations to increase the locality of memory references have been applied to many methods from numerical linear algebra including factorization methods like LU, QR and Cholesky [5] and iterative methods like 2D Jacobi [8] and multi-grid methods [14]. Many popular scientific libraries like LAPACK [1] are based on the BLAS (Basic Linear Algebra Subprograms), which can be considered as a de facto standard for the formulation of vector and matrix based numerical algorithms. The BLAS themselves are just a specification of the syntax and semantics of the operations, but many computer vendors provide efficient implementations of the BLAS for specific machines, in particular making effective use of the memory hierarchy of the machine.

Based on BLAS, there are efforts like PHiPAC (Portable High Performance ANSI C) [3] and ATLAS (Automatically Tuned Linear Algebra Software) [16] to provide efficient implementations of BLAS routines. ATLAS for example aims at the automatic generation of efficient BLAS routines by providing a code generator for the automatic creation of optimized on-chip (L1 cache) BLAS operations for specific platforms. The code generator determines the optimal blocking and loop unrolling factors by timings on the specific architecture. BLAS operations for larger arrays or matrices are built up from the fixed-size on-chip operations by architecture-independent code which partitions the matrix or vector operands into blocks of the given fixed size and arranges the computations such that L2 cache usage is optimized. PHiPAC takes a similar approach as ATLAS, but instead of forcing all problems to an independently optimized fixed format, PHiPAC directly optimizes each individual operation.

There are also approaches for other dense linear algebra algorithms or grid based methods [5, 8] but not for ODE solvers. The cache performance of two- and three-dimensional multigrid algorithms is investigated in [15]. In particular, a 2D red-black Gauss-Seidel relaxation method as the most time-consuming part of a multigrid method is considered and a blocking technique and array padding are applied to reorder the data accesses for increasing the temporal locality. The

increased data locality leads to significantly larger MFLOPS rates especially for grids with a fine discretization and many grid points. Software pipelining in general has been considered in [9].

6. Conclusions

We have addressed the efficient implementation of explicit RK methods for ODE systems arising from applying the method of lines to time-dependent PDEs. We have used the 2D-Brusselator equation as a typical example. Two different storage schemes for the Brusselator system, a row-oriented and a mixed row-oriented storage scheme, have been considered. The mixed row-oriented storage scheme has been used to derive a blockwise computation scheme that computes the blocks of the argument vectors in a pipelined way.

Runtime experiments have shown that for general RK implementations the mixed row-oriented storage scheme outperforms the row-oriented scheme on most of the processors considered. These results are due to higher locality caused by the smaller distance of the components accessed in one evaluation of the right hand side function $\mathbf{f}$. Because of the increase in locality obtained by the pipelining computation scheme, we have measured reductions in execution time between 10 % and 41 %.

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J.Ðu Croz, A. Greenbaum, S. Hammarlin, A. McKenney, and D. Sorensen. *LAPACK Users' Guide, Third Edition.* SIAM, 1999.

[2] R. Berrendorf and B. Mohr. *PCL - The Performance Counter Library, Version 2.0.* Research Centre Jülich, September 2000.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM Int. Conf. on Supercomputing*, 1997.

[4] R. W. Brankin, I. Gladwell, and L. F. Shampine. *RKSUITE release 1.0*, 1991.

[5] J. Choi, J. J. Dongarra, L. S. Ostrouchov, A. P. Petitet, D. W. Walker, and R. C. Whaley. Design and implementation of the ScaLAPACK LU, QR and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[6] W.H. Enright, D.J. Higham, B. Owren, and Ph.W. Sharp. A survey of the explicit Runge-Kutta method. Technical Report 94-291, University of Toronto, Department of Computer Science, 1995.

[7] E. Fehlberg. Classical fifth–, sixth–, seventh– and eighth order Runge–Kutta formulas with step size control. *Computing*, 4:93–106, 1969.

[8] K. S. Gatlin and L. Carter. Architecture-cognizant divide and conquer algorithms. In *Proc. of Supercomputing'99 Conference*, 1999.

[9] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes.* SIAM, 2001.

[10] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer–Verlag, Berlin, 1993.

[11] P. J. Prince and J. R. Dormand. High order embedded Runge-Kutta formulae. *J. Comp. Appl. Math.*, 7(1):67–75, 1981.

[12] T. Rauber and G. Rünger. Parallel Execution of Embedded and Iterated Runge–Kutta Methods. *Concurrency: Practice and Experience*, 11(7):367–385, 1999.

[13] T. Rauber and G. Rünger. Optimizing Locality for ODE Solvers. In *Proceedings of the 15th ACM International Conference on Supercomputing*, pages 123–132. ACM Press, 2001.

[14] L. Stals and U. Rüde. Data local iterative methods for the efficient solution of partial differential equations. In *Proc. of Computational Techniques and Applications*, 1997.

[15] C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory characteristics of iterative methods. In *Proceedings of the ACM/IEEE SC99 Conference*, Portland, Oregon, November 1999.

[16] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical report, University of Tennessee, 1999.

PERFORMANCE EVALUATION OF HYBRID PARALLEL PROGRAMMING PARADIGMS

Achal Prabhakar
Performance and Architecture Lab, CCS-3, LANL, New Mexico, USA
and
Department of Computer Science, University of Houston, Texas, USA
achal@cs.uh.edu

Vladimir Getov
Performance and Architecture Lab, CCS-3, LANL, New Mexico, USA
and
School of Computer Science, University of Westminster, London, UK
vgetov@lanl.gov

Abstract With the trend in the supercomputing world shifting from homogeneous machine architectures to hybrid clusters of SMP nodes, the interoperabiility of OpenMP and MPI has become a key issue in understanding and optimizing the overall system performance. While the low-level performance of MPI and OpenMP can be evaluated using existing benchmarks, the combination of the two poses new challenges. Therefore, a performance study of different hybrid programming paradigms is of high benefit for both the vendors and the user community. As part of our project, we have identified several possible combinations of the two models in order to provide qualitative and quantitative justification of situations in which any one of them is to be favoured. Collective operations are particularly important to analyze and evaluate on a hybrid platform and therefore we concentrate our study on three of them – barrier, all-to-all, and all-reduce. Issues like the optimal mix of OpenMP and MPI, the most efficient way of managing MPI communication from within OpenMP, the optimal unit of communication, and the degree of overlap between computation and communication need to be evaluated. The performance results supporting this investigation were taken on the IBM Power-3 machine at San Diego Supecomputer Center using our suite of hybrid microbenchmarks.

Keywords: performance evaluation, hybrid programming paradigms, parallel programming, clusters of SMPs, collective operations

1. Introduction

Originally, the single-program-multiple-data (SPMD) programming model was introduced to match distributed memory parallel machines with explicit message passing in the 80s [2]. Since then, the evolution of high-end computer architectures has seen the introduction of deeper and deeper memory hierarchies as well as the increasing heterogeneity of recent parallel platforms such as clusters of SMPs. Of course, one of the main motivations behind this continuously growing complexity in high-end architectures is the strive for higher performance. Indeed, the memory hierarchy concept mitigates the price/performance obstacles imposed by the existing memory technologies, while the adoption of hybrid architectures improves the scalability properties of modern high-end computer systems. However, one of the negative effects resulting from this process is the continuously widening performance gap when mapping the SPMD programming model onto the underlying parallel architectures.

Arguably, the most serious obstacle to the acceptance of parallel computing is the so-called *software crisis*. Software, in general, is considered the most complex artifact in computer science [5]. Since the lifespan of parallel machines has been so brief, their software environments rarely reach maturity and the parallel software crisis is especially acute. Hence, portability, in particular, is a critical issue in enabling high-performance parallel computing.

Two main programming models can be used within a single SMP node. The first one, the *true-SMP* programming model, assumes a single chunk of data for the whole SMP node and a corresponding program that allows (maybe at specific time-periods only) simultaneous processing while also accessing usually different parts of the local partition. This is usually achieved when the parallelisation of a program is done at loop level. In this case, most of the data should be considered as shared, while individual threads would normally have a relatively small amount of local variables. However, this is normally slow.

The second option is to apply the *SPMD* programming model inside an SMP node. This requires data partitioning between different threads within a single node. In this case, each thread has its own local data partition, while intra-node communications make use of shared-memory buffers located in a shared data segment within the node memory. This is usually faster than the true-SMP model.

In both cases above, correct and robust programming and execution can be achieved by appropriate use of the corresponding shared-memory programming constructs when accessing memory from the shared data segment(s). Also, it is usually fair to assume a one-to-one mapping between threads and processors for high-performance computing applications.

The trade-off between the two main programming models as described above depends on the frequency of memory references and their order in time for the

first approach, as well as the size of the "communication" buffers allocated in the shared-memory segment for the second one. In general, the overhead introduced by the intra-node communication using shared-memory buffers is usually smaller than the synchronization and coordination overhead of the trueSMP programming model [6]. Taking into account the above considerations, an application programmer still faces a number of choices that can change significantly both the overall performance and the programming style of a hybrid application code [11]. Therefore, we are going to consider several versions of the hybrid SPMD programming model for clusters of SMP nodes.

2. Hybrid Programming Models

Following the SPMD programming style, one can recognize several versions of the hybrid programming model which mostly refer to the way in which the inter-node communications are organized. The choices here are between the *master/slave* and the *workers farm* programming models on the one hand as well as with respect to the number of communication-only threads on the other hand. The number of simultaneous communication channels per SMP node that the system provides is also important for analyzing the performance trade-offs and identifying the best version of the hybrid programming model for a given application code.

Master/slave programming model. In this case, two specialized groups of threads are responsible for the execution – calculation threads (normally following one-to-one mapping between threads and processors) and the number of communication threads being between 1 and the number of simultaneous communication channels per node. A single master thread is responsible for thread creation and termination of the worker (calculation) threads. During processing the master thread is available and used for the inter-node communications. This model seems to be quite suitable for SMP nodes that provide only one external communication channel. Several communication threads (one of them being the master) are available for communication operations only, while a number of calculation threads perform the arithmetic work only.

Workers farm programming model. In this model threads are in charge of both calculation and communication which makes it very similar to the way many SPMD codes currently work. A one-to-one mapping between threads and processors seems the most natural in this case while message-passing operations will have to compete for communication resources. This model is very suitable for existing parallel applications with explicit message passing. It can also be used directly if SMP-aware MPI libraries are available on the target platform [1].

Some of the trade-offs to be considered in our work include the optimal mix of OpenMP and MPI, the most efficient way of managing MPI communication from OpenMP, and the overlap between computation and communication.

3. Evaluation Methodology

In order to evaluate the performance of hybrid OpenMP-MPI, we have selected three MPI global comunication primitives. These operations are mimicked in the SMP nodes using the corresponding OpenMP constructs. If there is no directly equivalent OpenMP construct corresponding to the MPI global primitive, the multithreaded part of the operation is implemented using the available OpenMP constructs [8]. Since the amount of work and the type of operation performed is same, the number of MPI processes and the number of OpenMP threads can be varied and the effects studied. Therefore, the implementation of our benchmarks is based on the extensible SKaMPI framework and allows great flexibility for tailoring the codes and planning the experiments [10].

For each test, there are four variations depending on which OpenMP thread performs the MPI communication. The simplest case is when the master thread performs communication along with computation, programmatically achieved using the OpenMP `master` directive. In cases when substantial load-imbalance among threads may be possible, better performance can be achieved if any thread that is available can perform the communication. This can be achieved in OpenMP by using either the `single` directive or explicitly using the lock/unlock primitives. The last case is when one thread is the dedicated communication thread and does not perform any computation. The test routine is written in the SPMD style with the master thread as the dedicated communication thread. For all sets of measurements the OpenMP-related workload is the same:

```
#pragma omp  for
    for (j = 0; j < schedule_loop_cap * omp_thread_count; j++)
      {
        omp_reduce_var += (work (work_param) == 0 ? 0 : 1);
      }
```

The extent of possible overlap [9, 4] of computation (OpenMP part) and communication would be greater if the threads did not have to synchronize. Thus, the measurements performed will illustrate the worst case overlap. This also implies that the measurements will include the slowdown due to load imbalances between the threads, which would be present otherwise also in a pure OpenMP implementation of the same code. As mentioned above, there are four ways of performing the synchronisation and communication.

The first one, master doing the communication, is classified in the *master/slave* programming model wherein a single master thread is responsible for the communication. This thread is however, also involved in the computation phase. In case a SPMD model is followed then the work could be distributed manualy and the master thread can be dedicated to communications only. This is the methodology followed for the dedicated communication thread case.

In the second method, any thread is allowed to perform communication using the `single` directive. It belongs to the *workers farm* programming model. The

thread which actually performs the communication is generally expected to be the first thread that completes the computation part. In the measurements, this is not really an issue as there is a barrier before any thread reaches the `single` directive. However, the measurements do emphasize the overheads due to the `single` directive itself. As noted in other OpenMP measurements, this overhead can be significant on some machines. This is also a worst case measurement as it compares the benefits of allowing any thread to communicate with the overheads of the OpenMP mechanism to implement that feature.

The third option is to use the explicit locking and unlocking facilities of OpenMP to synchronise. Each thread tries to `test_and_set` a lock and which ever one succeeds first, performs the communication while the rest wait at a barrier. The thread holding the lock performs the communication and releases the lock after completion of the communication and the barrier. Being similar to the single thread case, this allows any thread to perform the communication, this is more of a comparison between the two approaches of either using a `single` directive or using lock/unlock. In scenarios where the threads need not wait for the communication to complete and when there is a significant chance of load imbalance between the threads, this is expected to yield better performance since there is no implied barrier as in when using `single` and allows any thread to communicate.

In case we have a work that is not dependent, that is when we do not need a barrier at the end of the work, then in that case the relative load-imbalance between the threads can be offset by overlapping that with the communication. Any thread that is least loaded finishes first and can go on with the communication phase. In such a scenario, maximum overlap should be possible in the case when we use locks to make sure only one thread communicates. In any case, the implementation using the `single` directive is expected to be the slowest, provided that the extra overhead introduced by it is not in some manner aiding in reducing the network congestion due to rapid messaging.

4. Basic Tests

4.1 Hybrid barrier operation

The hybrid barrier measurement calculates the overhead of the combined OpenMP-MPI barrier. The most obvious manner to do this would be to have all the threads wait at a barrier, then letting one thread perform the communication while the rest wait at the subsequent barrier [3]:

```
#pragma omp barrier

#pragma omp master
MPI_Barrier(MPI_COMM_WORLD);

#pragma omp barrier
```

However, looking at the above, the first barrier appears to be redundant. The goal is not to synchronise all the threads in all MPI processes as that is least meaningful. Instead, one needs to synchronize each MPI process with the others using the MPI barrier with the task of synchronizing the OpenMP threads left to the individual MPI process [7].

Therefore, our benchmarking code eliminates the first barrier. Since the measurement is repeated a number of times, introducing some computation also prevents possible network congestion due to repeated MPI calls:

```
int mixed_omp_mpi_barrier (MPI_Comm comm)
{
 dummy_work_routine();

#pragma omp master
  gretval = MPI_Barrier (comm);

#pragma omp barrier

  return gretval;
}
```

The measurement is timed by the master thread on the root MPI process. It is repeated untill the standard deviation is lower than the user specified value or the experiment has been repeated for the maximum allowed number of times.

There are two different kinds of overlap possible between the computation and communication stages. Since the work routine is executed by all the threads, it might result in load imbalance causing some threads to arrive at the communication stage earlier than others. In case the concerned thread is also the communicating thread, the load imbalance will be compensated for by the overlap with communication.

While the above depends on the load imbalance, the overlap between the MPI communication and the subsequent OpenMP barrier would always be present. As shown in Figure 1, the hybrid barrier is a two stage process. The non-communicating threads complete the first stage of arrival at the barrier, while the communication is being performed by the communicating thread. The extent of overlap will vary depending on the implementation of the OpenMP barrier and the relative overhead of the first and second stage.

In case the work routine executes an OpenMP barrier at the end of the computation, as would happen in an OpenMP `parallel for reduction` construct, there would not be any overlap with communication and the load imbalance, if any, would only be offset by the overlap with the arrival stage of the OpenMP barrier.

As shown in the code section above, it is the master thread that does the communication. This however, may not always be the most optimal manner. There are cases where it is much more beneficial to allow any thread to perform the

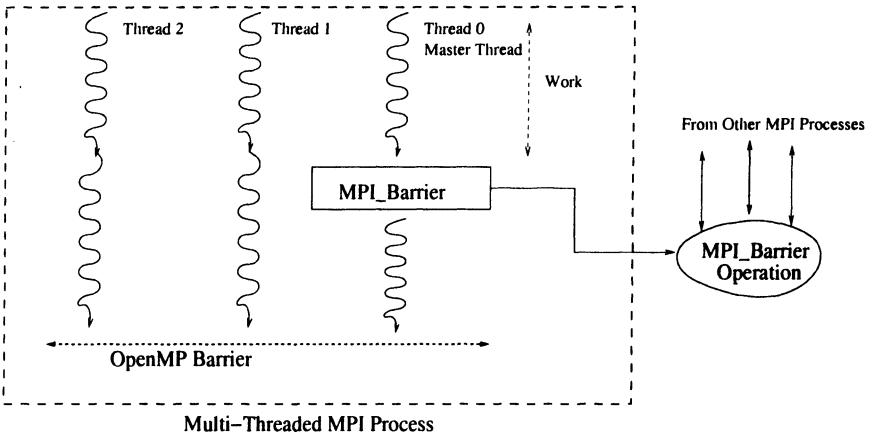


Figure 1. Hybrid barrier block diagram.

communication. For instance, if the communication is preceded by a section of code that does not define any variable used by the communication stage, then from the point of taking advantage of possible load-imbalance, it is advantageous that any thread which first finishes the work be allowed to perform the communication.

The preferred way to implement the - any one thread - logic in OpenMP is to use the `single` directive. It guarantees that any one and only one thread would execute the code contained in the single block. However, the standard also mentions that the `single` directive has an implied `barrier`, which is not desired as it can bring down the performance. The measurement code, using the `single` directive is:

```
int mixed_omp_mpi_barrier_single (MPI_Comm comm)
{

dummy_work_routine();

#pragma omp single
    gretval = MPI_Barrier (comm);
}
```

The only difference between the overhead using the `single` instead of the `master` directive thus is the presence of the `single` directive. Since we can find that from the individual OpenMP benchmarks, a conclusion can readily be made regarding the (dis)advantages of using `single`. It also makes calculating the extent of overlap achieved more easy and understandable.

The other method to implement the same scheme as with `single`, is to make use of the lock and unlock primitive provided by the OpenMP standard. The argument for using locks instead of `single` is very strong as on some machines, the lock/unlock primitives are considerably faster than the implementation of

`single` directive. Also, there is no implied barrier if locks are used. We use the `omp_test_set_lock` primitive to ensure that only one thread is holding the lock. Which ever thread manages to reach the communication section first tests and grabs the lock and continues on to perform the communication. Other threads, find the lock unavailable and enter the OpenMP barrier arrival stage. The thread which held the lock, also enters the barrier after completion of communciation and releases the lock at barrier departure. The code section describing this is:

```
do
{
  comm_done = -1;
  dummy_work_routine();
  mixed_omp_mpi_barrier_lock();
}while( more_work_required)

int mixed_omp_mpi_barrier_lock (MPI_Comm comm)
{
  int omp_thread_id;

  omp_thread_id = omp_get_thread_num ();

if ( comm_done == -1)
    if (omp_test_lock (&omp_lock) != 0)
      {
            comm_done = omp_thread_id;
           gretval = MPI_Barrier (comm);

      }
#pragma omp barrier
if (comm_done == omp_thread_id)
 {
  comm_done = -1;
  omp_unset_lock (&omp_lock);
 }

return gretval;
}
```

4.2 Hybrid all-reduce operation

The hybrid OpenMP-MPI all-reduce measurement evaluates the overhead of performing a global reduction operation across all threads in all MPI processes. Since the same collective all-reduce operation is performed both within as well as between the SMP nodes, the number of threads per MPI process and the number of MPI processes can be adjusted, keeping a constant number of overall processing entities. The results provide us with quantitative basis to determine the optimal mix of OpenMP threads per MPI-process and the total number of such MPI processes.

The hybrid all-reduce oepration is illustrated in Figure 2. As shown in the diagram, there are two distinct phases of the measurement. The first phase

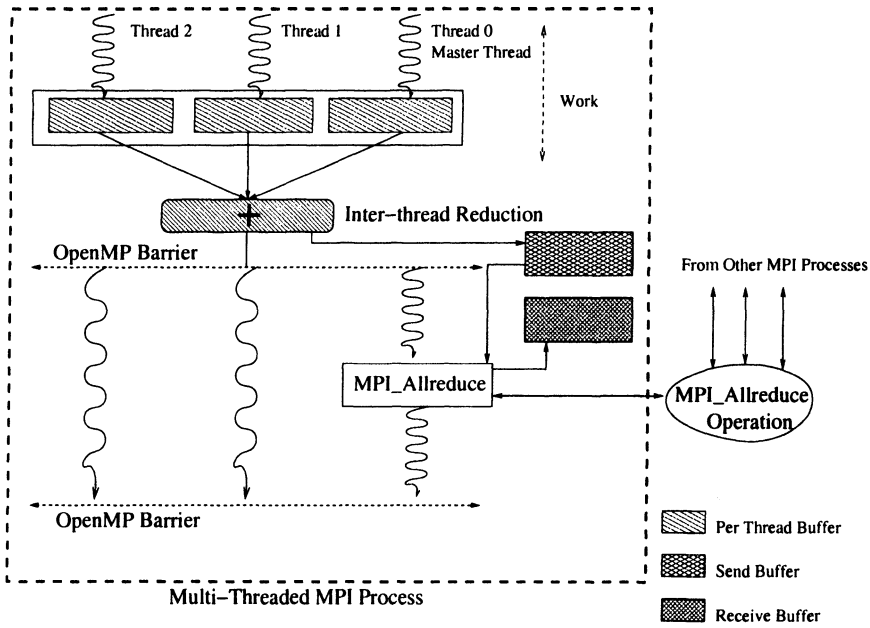


Figure 2. Hybrid all-reduce block diagram.

consists of the all-reduce operation is the intra-thread reduction being performed within the SMP node. Each thread allocates a private data buffer which it uses for storing the results of the computations assigned to itself. A shared buffer used for performing the intra-thread reduction if also allocated. After the completion of their respective computations, each threads private buffer has the data to be reduced. At this point, all the threads take turns to reduce their private data into the shared reduction buffer. To ensure that all threads are synchronised, this is followed by an OpenMP barrier. Since each thread needs to perform the reduction operation on the shared buffer, mutual exclusion is provided by using the OpenMP lock/unlock primitives. The second phase is the inter-node `MPI_Allreduce` operation, wherein each MPI process takes its locally reduced data and performs the global all-reduce operation. The shared buffer containing the intra-node reduced data is passed to the `MPI-Allreduce` primitive along with the receive buffer. After completion of this collecive operation, the recieving buffers on all participating nodes contain the globally reduced data. As the receiving buffer is shared, it is accessible to all the threads in a process. Thus after the second OpenMP barrier, all threads in every MPI process get the globally reduced data in the receiving buffer.

As mentioned previously, there are more than one methods of performing an MPI communication from within OpenMP. For the hybrid all-reduce measurement. The suite contains four different tests, each differing in the technique used

for deciding the threads that performs the MPI communication. The diagram illustrates the case when the master thread not only performs the computations but is also the designated communication thread. In the other case where the master thread is used for communication, it acts as a dedicated communication-only thread and does not take part in the computational phase. The other two cases are when none of the threads is designated as communication thread and the decision is taken at runtime on a first come basis. The only difference between these two cases is that while one uses the OpenMP `single` directive to enforce the any-one-thread rule, the other relies on explicit locking/unlocking. Also, since the `single` construct has an implied barrier at the end, the second OpenMP barrier immediately following the MPI communication is redundant and is not present for the test using the `single` directive.

```
void
measure_mixed_omp_mpi_allreduce (int len, MPI_Comm communicator)
{
  int my_id,lb,ub;

  my_id = omp_get_thread_num();

  lb = (len)*my_id;
  ub = (len)*(my_id +1);

#pragma omp  for
    for (j = 0; j < schedule_loop_cap * sphinx_omp_thread_count; j++)
      {
       omp_reduce_var += work(work_param);
      }

/* This is simulating the buffer that is a result of the previous computation */
        for(i=lb;i<ub;i++)
        hybrid_allreduce_buf[i] = (char)sphinx_omp_reduce_var;

/*Perform the reduction operation between threads - all threads do this*/
omp_set_lock(&omp_lock);
for(i=lb,j=0;i<ub,j<len;i++,j++)
         hybrid_allreduce_send_buf[j] |= hybrid_allreduce_buf[i];
omp_unset_lock(&omp_lock);
/* wait for all threads to perform the reduction */
#pragma omp barrier

 /* perform the reduction across the mpi processes */
#pragma omp master
 {
   gretval = MPI_Allreduce (hybrid_allreduce_send_buf,
             hybrid_allreduce_recv_buf, len, MPI_BYTE, MPI_BOR, communicator);
 }

#pragma omp barrier
}
```

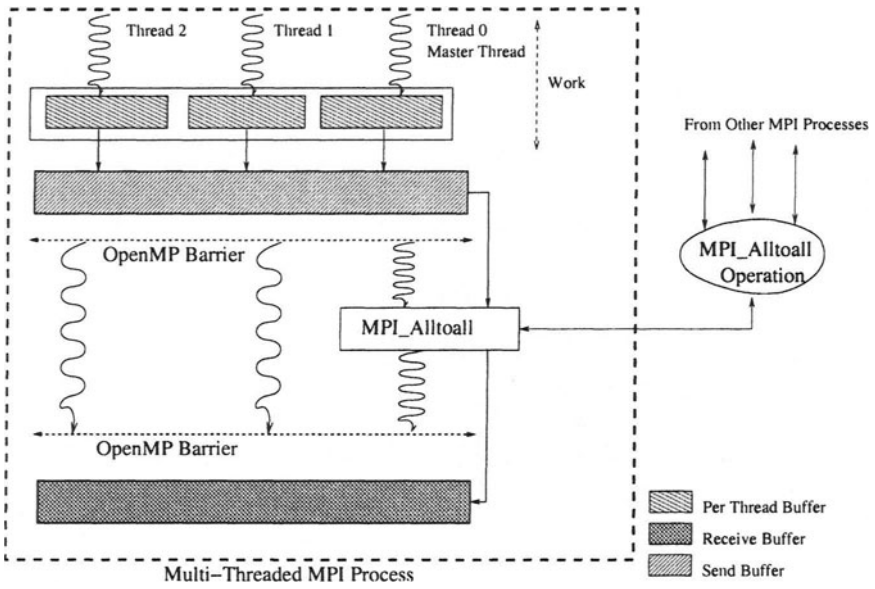


Figure 3. Block diagram for the hybrid all-to-all and all-gather operations.

4.3 Hybrid all-to-all operation

The hybrid OpenMP-MPI all-to-all measurement evaluates the overhead of performing a global collective communication among all the threads in all the MPI processes. The objective is to identify the potential performance advantages possible with delegating work to threads and to find the optimal number of OpenMP threads vs. the number of MPI processes. Since the `MPI_Alltoall` operation is highly communication intensive, a fair amount of performance enhancement is expected from a mixed mode implementation as opposed to a purely MPI implementation.

The hybrid all-to-all operation is illustrated in Figure 3. As shown in the diagram, there are two distinct phases of the measurement. The first phase consists of the inter-thread communication within the SMP node. Each thread is allocated a part of the computation work, the result of which is a data buffer private to the thread. The work is distributed using the OpenMP for worksharing directive. After completion of the computation, each thread copies the contents of its private buffer to a designated area within a shared buffer and subsequently waits at the OpenMP barrier for other threads to do the same. Since each thread is writing in a distinct portion of the shared buffer, there is no need for mutual exclusion in this case as opposed to the all-reduce measurement. After the completion of the barrier, each threads private data has been copied to the shared buffer, effectively allowing each thread to access the data from other

threads. This is equivalent to an all-to-all operation performed using shared buffers.

The second phase is the inter-node `MPI_Alltoall` operation, wherein each MPI thread communicates its buffer contents to other MPI processes and receives the same from all others. The shared buffer from the inter-thread operation is passed to the `MPI_Alltoall` routine by the communicating thread along with a shared receive buffer. After the completion of the all-to-all operation, the receive buffer contains data received from all MPI processes. Since this is a shared buffer, all threads have access to this data, thus completing the global all-to-all operation.

As mentioned in Subsection 4.2, there are more than one methods of performing an MPI communication from within OpenMP. Similarly for the hybrid all-to-all measurement, the suite contains four different tests, each differing in the technique used for deciding the thread that performs the MPI communication. In two of the cases, the master thread is the communicating thread. The difference being the two is that in one case the master thread is the dedicated communication thread and does not perform any computation while in the other case it performs computation as well as communication.

```
void
measure_mixed_omp_mpi_alltoall (int len, MPI_Comm communicator)
{
  int my_id,ub,lb;

my_id = omp_get_thread_num();

  lb = (len)*my_id;
  ub = (len)*(my_id +1);
#pragma omp  for
  for (j = 0; j < schedule_loop_cap * sphinx_omp_thread_count; j++)
    {
      omp_reduce_var += (work (work_param) == 0 ? 0 : 1);
    }
 /* Fill up the shared buffer */
 for(i=lb;i<ub;i++)
   hybrid_alltoall_buf[i] = (char)omp_reduce_var;

#pragma omp barrier
#pragma omp master
{
  gretval = MPI_Alltoall (hybrid_alltoall_buf,send_count, MPI_CHAR,
            hybrid_alltoall_recv_buf, recv_count, MPI_CHAR, communicator);
}

#pragma omp barrier

}
```

The other two cases are when none of the threads is designated as communication thread and the decision is taken at runtime on a first come basis. The only difference between these two cases is that while one of them uses the

OpenMP `single` directive to enforce the any-one-thread rule, the other relies on explicit locking/unlocking. Also, since the `single` construct has an implied barrier at the end, the second OpenMP barrier immediately following the MPI communication is redundant and is not present for the version using the `single` primitive.

In the all-gather operation each MPI node collects data from all the other MPI nodes into a contiguous buffer. Extension of this to the hybrid scenario requires that the same operation is mimicked within the SMP node as an inter-thread all-gather operation. Similar to the hybrid all-to-all, the objective is to identify the potential performance advantages possible with delegating work to threads and to find the optimal number of OpenMP threads vs. the number of MPI processes. However, the all-gather is less communication intensive when compared to the all-to-all operation as there is no scatter operation among the MPI processes. The two tests differ in the MPI routine with much of the inter-thread OpenMP part remaining the same. Therefore, the block diagram for the hybrid all-gather operation is illustrated in the same Figure 3 as the all-to-all operation. As shown in the diagram, the two distinct phases of the measurement are similar to the hybrid all-to-all measurement with the difference being only in the MPI routine used. As such, the description from the previous Subsection apply to this measurement as well.

The suite consists of four separate tests for the hybrid all-gather measurement, each emphasizing a unique way of using OpenMP with MPI. Two of the four cases use the master thread for communication, with one written in the SPMD style so as to dedicate the master thread to communication and does not allocate any work to it. The other two let any thread perform the communication using the OpenMP `single` and the lock/unlock primitives.

5. Results and Discussion

The experiments were conducted on an IBM SP3 8-way SMP with 375 MHz Power3 processors, using the VisualAge C++ Professional or C for AIX compiler, version 5. All the times mentioned in the results are in microseconds.

The results for the hybrid OpenMP-MPI barrier measurement are depicted in Figures 4, 5, and 6 for the three cases with master thread commnicating, using lock/unlock and using the `single` directive respectively. The graphs show that there is a significant overlap between the MPI communication and the OpenMP barrier operation.

The plot for a single MPI process shows a steep positive gradient, especially when going from 7 to 8 threads, which is caused by the increased overhead of the OpenMP barrier operation as the number of threads increases. The sharp jump from 7 to 8 threads can also be attributed to the fact that the machine has 8 processing elements which might lead to contention between the system

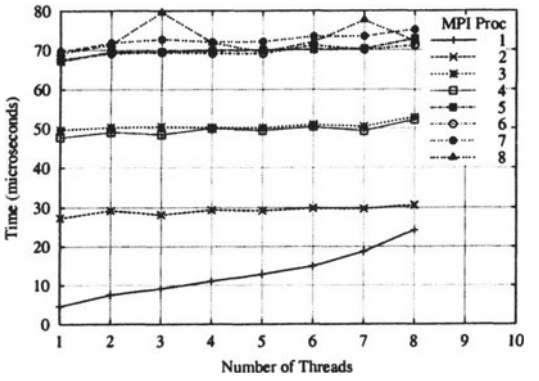


Figure 4. Hybrid barrier: Using master thread communicating.

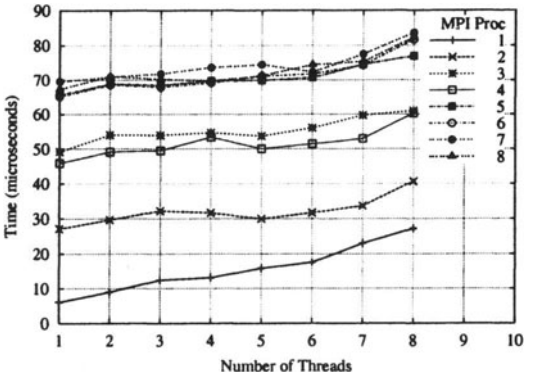


Figure 5. Hybrid barrier: Using OpenMP lock/unlock for synchronisation.

threads and the user level threads. However, with more than 1 MPI process the curve is almost flat illustrating that almost all of the cost of the OpenMP barrier is being absorbed via overlap in the MPI communication costs. The abnormal jump for 3 and 8 threads and 8 MPI processes seems more like an outlier caused by system load factors rather than as a trend as the same is not apprarent when using locks (see Figure 5) or the `single` primitive (see Figure 6) as well as with any other curve when using the master thread.

The trend is similar for the other two cases except that due to increased overheads of using locks or the `single` directive, the overall cost of the construct increases. The increase is most noticeable going from 7 to 8 threads as the OpenMP threads need to perform more computation (sleep, wait) and thus compete with the system threads. In all the three cases, the overhead of the mixed barrier for 4 MPI processes is lower than that for 3 MPI processes; the

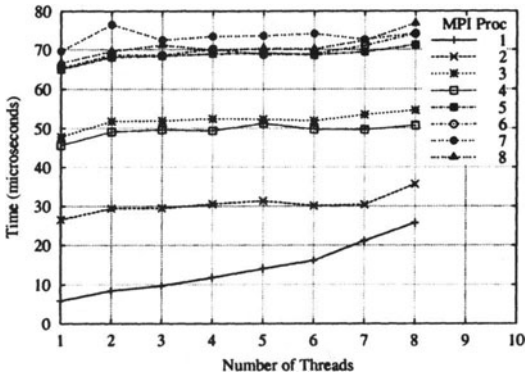


Figure 6. Hybrid barrier: Using the OpenMP `single` directive for synchronisation.

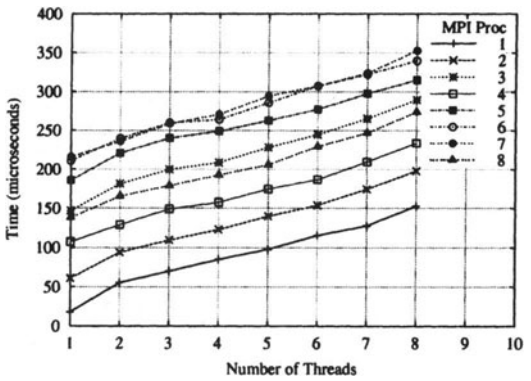


Figure 7. Hybrid all-reduce: Using master thread communicating.

same being true for 8 and 7 MPI processes in general, discounting the outliers in Figure 4.

Overall, the best performance is achieved when using the OpenMP `master` directive to allow only the master thread to communicate. The overhead when using lock/unlock is almost the same as when using the `single` directive, although this case performs a bit better than lock.

The results for the hybrid all-reduce operation are plotted in Figures 7, 8, 9, and 10 for the four cases using master for communication, using lock/unlock, using the `single` directive and using the master thread as the dedicated communication thread respectively.

The most significant observation made from the four graphs is that the overhead of the hybrid all-reduce operation with 4 and 8 MPI processes is lower than that for 3, 5, 6, or 7 MPI processes. This points to an alternate MPI all-reduce

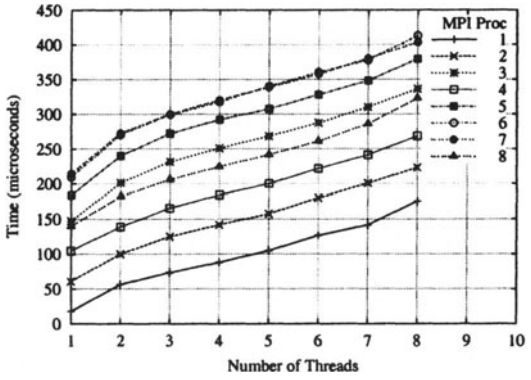


Figure 8. Hybrid all-reduce: Using OpenMP lock/unlock for synchronisation.

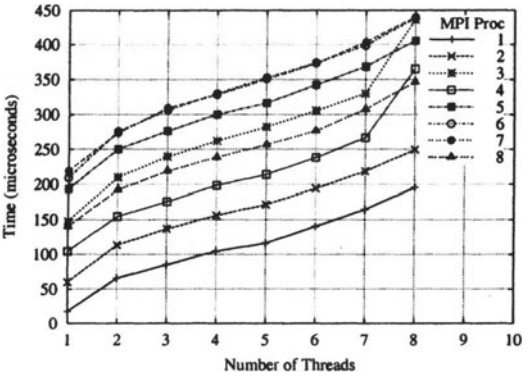


Figure 9. Hybrid all-reduce: Using the OpenMP `single` directive for synchronisation.

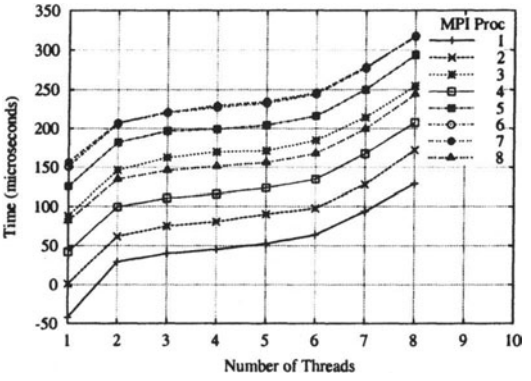


Figure 10. Hybrid all-reduce: Using dedicated communication thread.

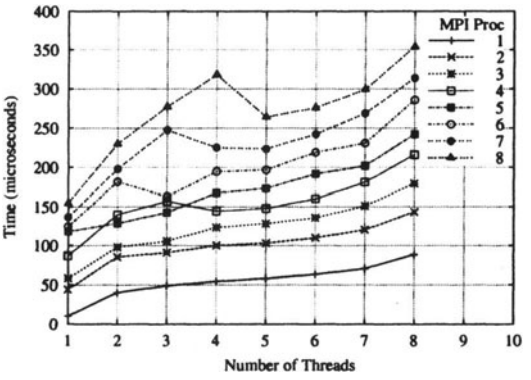


Figure 11. Hybrid all-to-all: Using master thread communicating.

strategy being used for 4 and 8 processes. As we were unable to conduct the experiment with more nodes we can not assert that this is true for other powers of two also.

For all the four cases, the overhead increases almost linearly with increasing number of threads, except when using OpenMP `single` there is a sharp jump when going from 7 to 8 threads with 3 and 4 MPI processes. This jump however is more likely a factor of system load rather than a characteristic as it is not apparent in the other three cases.

The overall cost of the operation is maximum when using OpenMP `single` followed closely by the lock/unlock method. A note about the graph in Figure 10 is due: since the master thread acts as the dedicated communication thread and does not take part in the computational phase, the effective number of threads should be read as one minus the corresponding x-axis value. With that fact, this last method causes the least overhead, with cost lower than when using the `master` directive as the disgnated communication thread and much lower than either the lock/unlock or the single thread case. This is possibly due to decreased contention between the system and user threads, as the master thread only communicates and thus does not compete for cpu time with the system threads. It also create increased possibility for the overlap of computations with the MPI commmunication phase.

The results for the hybrid all-to-all measurement are plotted in Figures 11, 12, 13, and 14 for the four cases using master for communication, using lock/unlock, using the `single` directive and using the master thread as the dedicated communication thread respectively.

The all-to-all results show a more expected behaviour with overhead increasing with increasing number of threads as well as with increasing number of MPI processes. With 8 MPI processes and 4 OpenMP threads, there is a

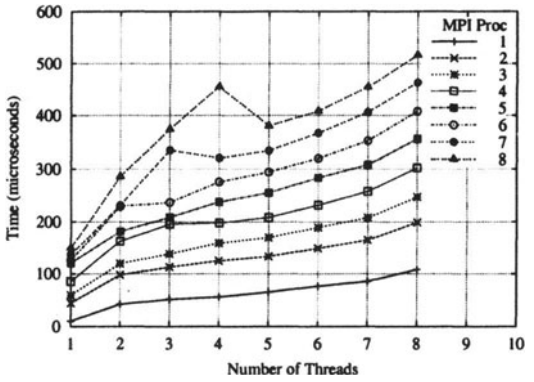


Figure 12. Hybrid all-to-all: Using OpenMP lock/unlock for synchronisation.

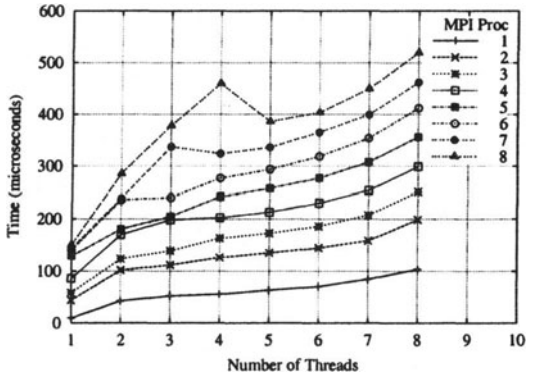


Figure 13. Hybrid all-to-all: Using the OpenMP `single` directive for synchronisation.

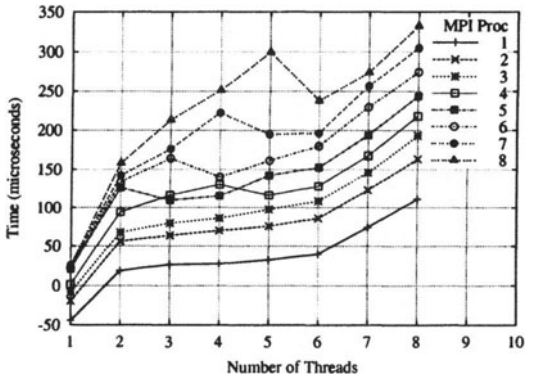


Figure 14. Hybrid all-to-all: Using a dedicated communication thread.

steep increase in cost for all the four cases. The overhead when using the lock/unlock primitives and the `single` directive is almost the same, and is significantly higher than when using the master as the designated communication thread. This indicates that there is no potential advantage in using either of the two as compared to the simpler method of using master. It also implies that there is little overlap between the execution of the OpenMP constructs and MPI communication.

Comparing the overhead for master thread as the dedicated versus the designated thread is more complex. In general, the overhead is lower with a dedicated thread when the number of threads is lesser than 5 and increases steeply when more than 5 threads are used.

The results for the hybrid all-gather measurement are not shown on separate figures in this paper but follow a trend similar to the all-to-all measurement. The maximum overhead is incurred when using either the lock/unlock or the `single` directive, suggesting that the any-thread-communicates paradigm is suitable only when the OpenMP work is fairly coarse grained and significant load imbalance between threads is expected. In scenarios where the computational phase is strictly synchronized, as it is when using `single`, the extra overhead of the `single` directive only adds to the total cost. As such it is much better to use the simpler – master thread as the designated communicating thread method. Otherwise, the performance of the `single` is almost the same as lock/unlock and thus is preferred as it is easier to use. However, the `single` directive has an implied barrier, which may not be needed in all scenarios in which case lock/unlock is expected to provide better performance.

The overhead of the dedicated master thread method as compared to the designated master thread method is better for small number of threads. As was the case with the all-to-all operation, the performace of this method decreases with increasing numeber of threads. Specifically, the overhead is greater than the designated thread method when the number of threads are greater than 5.

6. Conclusions

We have presented an enhanced set of benchmarks for evaluating the performance of hybrid OpenMP-MPI constructs with focus on three collective operations – barrier, all-to-all, and all-reduce. The implementation of our benchmarks is based on the extensible SKaMPI framework and allows great flexibility for tailoring the benchmark both externally and internally. We have shown and summarized the performance measurements for the IBM SP3 installation at the San Diego Supercomputing Center. Critical discussion of the results has also been presented highlighting the optimal mix of OpenMP and MPI, the most efficient way of managing MPI communication from within OpenMP, the op-

timal unit of communication, and the degree of overlap between computation and communication.

Acknowledgments

We convey our sincere thanks to the staff at the San Diego Supercomputing Center for the use of their computing facilities and for the technical support while running the experiments presented in this paper.

References

[1] A. Chowdappa, A. Skjellum, and N. Doss. *Thread-Safe Message Passing with P4 and MPI*, Technical Report TR-CS-941025, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1994.

[2] F. Darema, D.A. George, V.A. Norton, and G.F. Pfister. A Single-Program-Multiple-Data Computational Model for EPEX/FORTRAN. *Parallel Computing*, 7(1):11-24, April 1988.

[3] B. de Supinski and B. Chan. Towards an Integrated Parallel Microbenchmark Suite. *Proc. Workshop on OpenMP Applications and Tools (WOMPAT)*, July 2000.

[4] T. Fahringer, M. Haines, and P. Mehrotra. On the Utility of Threads for Data Parallel Programming. *Proc. of the 9th ICS*, pages 51-59, July 1995.

[5] E.A. Feigenbaum. The Intelligent Use of Machine Intelligence. *CrossTalk – The Journal of Defense Software Engineering*, http://www.stsc.hill.af.mil/crosstalk/1995/08/index.html, 8(8):10-13, August 1995.

[6] G. Krawezik, G. Alléon, F. Cappello. SPMD OpenMP versus MPI on a IBM SMP for 3 Kernels of the NAS Benchmarks. *Proc. of the ISHPC-IV Symposium, LNCS*, 2327:425-436, Springer-Verlag, 2002.

[7] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, July 1978.

[8] OpenMP Architecture Review Board. *OpenMP Fortran Application Program Interface (Version 1.1)*. Technical Report, http://www.openmp.org/specs/mp-documents/fspec11.pdf, November 1999.

[9] M.J. Quinn and P.J. Hatcher. On the Utility of Communication-Computation Overlap in Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 33(2):197-204, March 1996.

[10] R.H. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A Detailed, Accurate MPI Benchmark. In *Proc. 5th EuroPVM-MPI Meeting, LNCS*, 1497:52-59, Springer-Verlag, 1998.

[11] L.A. Smith and P. Kent. Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code. *Proc. of the 1st European Workshop on OpenMP*, pages 6-9, Sept. 1999.

PERFORMANCE MODELLING FOR TASK-PARALLEL PROGRAMS

Matthias Kühnemann
Department of Computer Science
Technical University of Chemnitz, Chemnitz, Germany
kumat@informatik.tu-chemnitz.de

Thomas Rauber
Department of Mathematics and Physics
University of Bayreuth, Bayreuth, Germany
rauber@uni-bayreuth.de

Gudula Rünger
Department of Computer Science
Technical University of Chemnitz, Chemnitz, Germany
ruenger@informatik.tu-chemnitz.de

Abstract Many applications from scientific computing and physical simulations can benefit from a mixed task and data parallel implementation on parallel machines with a distributed memory organization, but it may also be the case that a pure data parallel implementation leads to faster execution times. Since the effort for writing a mixed task and data parallel implementation is large, it would be useful to have an a priori estimation of the possible benefits of such an implementation on a given parallel machine. In this article, we propose an estimation method for the execution time that is based on the modelling of computation and communication times by runtime formulas. The effect of concurrent message transmissions is captured by a contention factor for the specific target machine. To demonstrate the usefulness of the approach, we consider a complex method for the solution of ordinary differential equations with a potential for a mixed task and data parallel execution. As distributed memory machine we consider the Cray T3E and a Linux cluster.

Keywords: execution time analysis, runtime formulas, mixed task and data parallelism, scientific computing

1. Introduction

We consider task parallel programs in a form where the entire program is built up from multi-processor tasks (M-task) each of which can be executed on an arbitrary number of processors and is often implemented in a data-parallel way. Different M-tasks can cooperate in a compositional way which means that one task produces data to be used as input data by another task. But a program may also contain independent tasks which can be executed concurrently to each other on subsets of processors, also called groups of processors. There usually exists a large variety of different parallel realizations of the same application program. The program versions can differ in the execution order of the tasks and the mapping of each M-task onto a set of processors of a specific size. In this paper we address the question how to select an efficient parallel program version from the described class of programs when using a distributed memory machine (DMM).

Whether a pure data parallel or a mixed task and data parallel program version is more efficient strongly depends on the specific application program and its requirement for communication, especially for collective communication operations. Collective communication operations performed on smaller processor groups lead to smaller execution times due to the logarithmic or linear dependence of the communication times on the number of processors [11, 9, 6]. This behavior of the communication time can be an advantage for a concurrent computations on disjoint subsets of processors. In general, the effort to implement a task parallel program is considerable and so it would be useful to have an effective method to determine the most efficient program version before implementing all the details. Such a method needs a notion of costs assigned to different program versions under consideration where the costs of a specific version meet some requirements: they should be based on the specific task structure and they should reflect the behavior of the parallel execution times resulting on a specific DMM.

In this paper, we suggest a method based on costs estimating parallel execution times of task parallel programs with M-tasks. The modelling of costs is based on a specification of the task structure and assumes a corresponding partition of the set of processors into groups or subsets of processors. The costs are given as runtime formulas that contain parameters describing the parallel target machine and characteristics of the algorithm to be implemented. Runtime formulas have been used before for modelling the execution time of communication operations in isolation [11, 9], but their use for modelling the execution time of task parallel programs has not yet been studied. The runtime formulas are built up from computation costs and communication costs which reflect the actual execution of a computation or a communication operation in isolation. But due to complex processor architectures, memory hierarchies and

network properties, the execution time of an entire program might strongly be influenced by cache effects or network contention. Our aim is to investigate whether a purely program-oriented construction of the runtime formulas is suitable to model costs for task parallel executions and whether additional effects like cache effects and network contention have to be taken into consideration to obtain an accurate prediction.

In the following sections, we describe how to build up program-specific runtime formulas for parallel platforms with distributed memory. In Section 2 we summarize the general approach. Section 3 briefly introduces the parallel target platforms used, a Cray T3E and a Beowulf cluster of PCs, and presents the runtime formulas for communication operations for those machines. As complex example applications we consider one-step methods for solving ordinary differentiation equations (ODEs) which have a potential for M-task parallelism. In Section 4, we model the runtime formula for an entire application. Section 5 shows that the runtime formulas reflect the performance behavior of the parallel program and Section 6 concludes.

2. Runtime formulas

The runtime formulas of our cost model are composed according to the M-task structure of the programs whose execution time they describe and summarized in the following. The runtime formulas consist of two parts describing the computation times and the communication times.

Computation and communication operations. The computation times are modelled by taking into account the number of operations to be executed. For each arithmetic operation we use a computation time t_{op} that can be determined by runtime measurements with simple sequential test programs on the parallel machine used. In addition, we use execution times T_f for specific functions f which are needed to describe specific application problems. The execution time for such a function f can be build up from single computation operations or can be a measured time for the function in isolation. The reason to use a single value T_f in the runtime formula is that most numerical methods, like ODE solvers, are designed as black-box solvers, so that they can be used for arbitrary functions f describing the right-hand side of the ODE system.

The communication times are modelled by formulas that describe the execution time of individual communication operations, such as single-transfer or broadcast operations. These formulas are used to model the internal communication time of a M-task, which are often realized in a data parallel way. Communication formulas may also be used to describe the time for data redistributions inserted between different multi-processor tasks related in a compositional way. The runtime formulas for communication operations are functions in closed form that depend on the message size b and the number p of pro-

cessors participating in the communication operation. For a given DMM, the value of these parameters result from modelling the runtime formulas by the least squares method. In Section 3 we present runtime formulas for different communication operations in isolation, see also [11, 9, 4].

Task parallel programs. Runtime formulas for entire programs are built up according to the task structure of the specific program. First, each participating M-task M is assigned a runtime formula built up from communication and computation times according to the internal structure. We assume a SPMD-like internal computation structure consisting of alternating phases of computation and communication, so that the execution time can be represented by

$$T_M(p, b) = T_{comp}(p, b) + T_{comm}(p, b).$$

$T_{comp}(p, b)$ is the maximum of the execution times of the participating processors. $T_{comm}(p, b)$ is the sum of the runtimes of the communication operations used. Second, the runtime of an entire program is built up from the costs of the M-tasks according to the task structure. In the case that two M-tasks M_1 and M_2 are executed concurrently to each other on disjoint sets of processors, the maximum of the runtimes is taken:

$$max(T_{M_1}(p_1, n_1), T_{M_2}(p_2, n_2)).$$

$T_{M_i}(p_i, n_i)$ denotes the runtime of task M_i for problem size n_i on a group of p_i processors, i=1,2. The sets of processors executing M_1 and M_2 are assumed to be disjoint. In the case that two M-tasks M_1 and M_2 are executed one after another and the runtimes are added:

$$(T_{M_1}(p, n_1) + T_{M_2}(p, n_2)) + T_{redist}(p, m).$$

Both times $T_{M_i}(p, n_i)$ contain the same parameter p for the sizes of the processor groups indicating that both tasks M_1 and M_2 are executed on a processor group of the same size, which is usually the same group. In addition, some communication between successive task activations might be necessary to realize redistribution of data. The time needed for the redistribution is denoted by $T_{redist}(p, m)$ where p is the number of participating processors and m the size of the data to be redistributed.

2.1 Comparison with other cost models

Our approach of performance modelling with runtime formulas combines a modelling part for SPMD-like computations and a construction part for the upper level task parallelism of M-tasks. The combination is inspired by the specific goal to compare the efficiency of different realizations with mixed task and data parallelism for the same given algorithm. Other cost models for

parallel programming, like the LogP model and its variations [2, 1] or the BSP model [7, 5], are aimed at aspects different from ours. The LogP model is more architecture oriented with an emphasis on a detailed modelling of parallel runtimes but is less suited to express explicitly upper level task parallelism. The BSP model concentrates on a specific program structure of supersteps which is not straightforwardly related to a mixed task and data parallelism with M-tasks. The advantage of using runtime formulas for the modelling of the execution times lies in the fact that they can easily be adapted to a mixed task and data parallel execution of program parts. Moreover, runtime formulas are able to capture the effect that the same communication operation may lead to different execution times, depending on whether other communication operations are simultaneously executed by concurrent processor groups or not. This effect cannot easily be captured by other models like the BSP model [7, 5] or the LogP model and its variations [2, 1] that assume a pure data-parallel execution of the program. In these models, it is difficult to capture runtime effects that occur only in a task parallel execution. Using runtime formulas, such effects can be captured, e.g., by contention factors introduced in Section 3.2.

In constrast, the presented approach of runtime formulas exactly matches this program structure and its realization with by separating the two programming levels also in the cost model, i.e. by combining different methods of runtime estimation for the different kinds of parallelism. computations as it will be discussed in the next section. The further construction of more complex runtime formulas exactly mimics the task structure to be considered. Altogether, this results in a method that is ideally suited to be used by application programmers to compare the expected parallel runtime of different task structures without dealing with low-level architectural and implementation details.

3. Modelling Communication Costs

We consider the communication on different machine with distributed memory, a Cray T3E-1200 and a Beowulf-Cluster. The T3E uses a three-dimensional torus network. The six communication links of each node are able to simultaneously support hardware transfer rates of 600 MB/s for the T3E.
The Beowulf Cluster CLiC ('**C**hemnitzer **Li**nux **C**luster') is build up of 528 Pentium III processors clocked at 800 MHz. The processors are connected by two different networks, the communication network and the service network. Both are based on the fast-Ethernet-standard, i.e., the PEs can swap 100 MBit per second. The service network (Extreme Block Diamand) allows external access to the cluster. The communication network (Cisco Catalyst) is used for inter-process communication between the PEs.

We consider message-passing programs that are coded using the MPI standard [3]. On the CLiC, LAM MPI, version 6.3.2 is used.

3.1 Communication operations in isolation

For single-transfer operations and collective communication operations we consider different variants and measure their runtime on the CLiC and on a Cray T3E-1200. For single-transfer operations, the runtimes are obtained on a 4-processor partition with processor 0 sending data to processor 3. The message sizes are between 2 KByte and 400 KByte. Except for some anomalies of the buffered single-to-single transfer on the Cray T3E, the runtimes increase linearly with the message size. On the Cray T3E the standard single-to-single transfer is the fastest operation. For the CLiC we have additionally implemented a piecewise single-to-single transfer operation that splits the message to be transmitted into pieces of 4KB and sends the pieces separately. This is the fastest single-transfer operation on the CLiC.

For collective communication operations we consider single-broadcast operations (`MPI_Bcast()`), accumulation operations (`MPI_Reduce()`), gather operations (`MPI_Gather()`), scatter operations (`MPI_Scatter()`), and multi-broadcast operations (`MPI_Allgather()`). The execution times for collective communication operations on the Cray are essentially faster than on the CLiC. Again, we have implemented piecewise communication operations on the CLiC that turn out to be faster than the original operations.

Runtime formulas: The runtime formulas that are used for the modelling are summarized in Table 1. The value b denotes the message size in bytes, p is the number of processors participating in the communication operation. The coefficients τ and t_c can be considered as startup and byte-transfer times and are determined by curve fitting with the least-squares method. For different internal realization of the same communication operation, different values for the coefficients are obtained. We therefore use an additional parameter V to distinguish the different variants; the specific values for the coefficients τ (V), t_c(V), t_1(V) and t_2(V) are given in Tables 2 - 3.

Table 1. Runtime formulas for communication operations.

Operation	*Runtime formula*
MPI_Send	$t_{s2s}(b) = \tau + t_c \cdot b$
MPI_Bcast	$t_{sb}(p,b) = \tau log_2(p) + t_c \cdot log_2(p) \cdot b$
MPI_Reduce	$t_{sa}(p,b) = \tau log_2(p) + t_c \cdot log_2(p) \cdot b$
MPI_Allgather	$t_{mb}(p,b) = \tau_1 + \tau_2 \cdot p + t_c \cdot p \cdot b$

Single transfer: The runtime for a single-to-single transfer is modelled by a linear function $t_{s2s}(b) = \tau + t_c \cdot b$ where $\tau(V)$ denotes a *startup time* and $t_c(V)$ denotes the *byte-transfer time* for the specific operation V. Curve fitting results

in the parameter values given in Table 2. The negative coefficients arise when using the least squares method for measured runtimes in the complete range of message sizes. Restricting the method to small message sizes up to 2 KByte leads to a separate, more accurate runtime formula for small messages without negative coefficients. A comparison between the predicted and measured runtimes for all MPI single-transfer operations shows that the predictions fit the measured runtimes quite accurately, especially for large messages (not shown in a figure).

Table 2. Parameter values for single-transfer operations and for collective operations such as single-broadcast and accumulation with a logarithmic dependence on the number of participating processors.

Operation	*CLiC*		*Cray T3E*	
variant	$\tau(V)[\mu s]$	$t_c(V)[\mu s]$	$\tau(V)[\mu s]$	$t_c(V)[\mu s]$
MPI_Send	-7.526	0.10607	13.965	0.00267
Send_P	145.021	0.08804	9.748	0.00533
MPI_Bcast	-3420.688	0.3409	7.723	0.0039
Bcast_P	564.125	0.0939	7.007	0.0050
MPI_Reduce	-119.871	0.1223	168.516	0.0093

Single broadcast and accumulation: The modelling of single-broadcast (`MPI_Bcast()`) operations, see Table 1, uses a logarithmic dependence on the number p of processors because the broadcast transmissions are based on broadcast trees with logarithmic depth. The same formula can also be used for the prediction of single-accumulation operations (`MPI_Reduce()`).

Multi-broadcast and gather/scatter: The runtime formula for multi-broadcast operations (`MPI_Allgather()`), see Table 1, increases linearly with the message size and the number of processors. This formula can also be used to model scatter and gather operations. The resulting coefficients are shown in Table 3.

3.2 Communication in task parallel executions

Until now, we have considered runtime formulas for the execution time of communication operations in isolation, which means that there was no concurrent communication operation of the same application program in transmission. In Section 4 those runtime formulas are used for the modelling and prediction of data parallel realizations of regular programs from scientific computing [9–10]. But it is not a priori clear whether those runtime formulas can also be used for modelling the execution time of mixed task and data parallel programs and in

Table 3. Parameter values for the runtime formulas with a linear dependence on the number of participating processors. MultiBcast_P and Allgather_P are piecewise implementations.

Operation	*CLiC [μs]*			*Cray T3E [μs]*		
variant	$\tau_1(V)$	$\tau_2(V)$	$t_c(V)$	$\tau_1(V)$	$\tau_2(V)$	$t_c(V)$
MultiBcast	-33270.8	21801.5	1.031	-3.72	42.60	0.028
MultiBcast_P	-9724.2	8385.5	0.690	-24.20	-0.81	0.036
MPI_Allgather	9175.3	-7542.0	3.182	6.04	-0.75	0.019
Allgather_P	-669.9	543.3	2.806	7.72	13.23	0.018
MPI_AllgatherV	-709.6	-957.0	5.443	11.83	17.47	0.019
MPI_Gather	-316.2	654.5	0.095	31.08	-118.51	0.006
MPI_Scatter	24.6	1439.5	0.086	-0.48	5.45	0.003

fact, experiments have shown that using the runtime formulas from Subsection 3.1 leads to predicted runtimes that are too small compared to the measured runtimes. The effect is much larger on the Beowulf cluster than on the T3E. This behavior indicates that concurrent communication operations of the same application program can interfere with each other in the sense that the transmission time is delayed, especially when many single-to-single transmissions are executed concurrently.

Further experiments have shown that only a slight change of the runtime formula for communication operations is needed to model the runtime of concurrent communication transmissions. The change can be concluded from the following observation made when using the runtime formulas for data parallel realizations from Subsection 3.1 to model the runtime for concurrent collective communication operations: The difference between the measured and the predicted runtimes is increasing with the number p of processors (used for the entire application program) and the size n of the message transmitted. The dependence on p is more significant than the dependence on n. Both dependencies correspond to the expectation that collisions in the network become more likely when the number of participating processors and the message sizes are increasing. To capture the interference of a concurrent execution of communication operations in the runtime formulas, we adjust the byte transfer time of the runtime formulas by introducing a network contention factor $C(p, n)$ that depends on p and n. In principle, this factor can be determined by a detailed queuing analysis, but this would require the knowledge of the internal realizations of the MPI communication operations. Thus, we take the approach to determine the contention factor empirically starting from the runtime estimation with the runtime formulas from Subsection 3.1.

We determine the contention factor by comparing the delay in the execution times for communication operations, when they are performed concurrently, using the execution times of these operations without network contention. The contention factor itself is modelled as a function of p and n and the shape of the contention factor (or contention function) has been determined by experiments from the measured delays in the execution times. The parameters within this function are then determined by curve fitting. The resulting contention factor is used for modelling the communication times of those program parts of complex applications that are executed in a mixed task and data parallel way. Each parallel machine is characterised by a different contention factor because of the different network architectures of the machines. To summarize, using the contention factor, the structure of the runtime formulas for one communication operation does not change but the contention factor is used to adjust the byte-transfer time t_c to the specific situation. Especially, the startup times are not changed. Thus, for example, the runtime formula of a multi-broadcast operation is

$$\tilde{t_{mb}}(p,b) = \tau_1 + \tau_2 \cdot p + C(p,n) \cdot t_c \cdot p \cdot b. \quad (1)$$

The same contention factor is used for each communication operation, i.e., there is no need to determine different contention factors for different communication operations. The communication operations of pure data parallel parts of application programs are modelled without the contention factor because there is no interference of message transmission. The contention factors for the modelling of concurrent message transmissions are

$$\begin{aligned} C_{T3E}(p,n) &= 0.04 \cdot p \cdot log_2(log_2(p)) \cdot log_2(n) \\ C_{CLiC}(p,n) &= 0.0045 \cdot p \cdot p \cdot log_2(p) \cdot (log_2(n) + log_2(p)), \end{aligned}$$

where p denotes the total number of processors participating in the execution and n the size of the message transmitted.

4. Example Application

To investigate the usefulness of the modelling approach for complex application programs, we consider parallel implementations of a specific solution method for ordinary differential equations (ODEs), the iterated Runge-Kutta method (iterated RK method).

4.1 Task structure of iterated RK methods

The iterated RK method is an explicit one-step method for the solution of initial value problems of ODEs. The iterated RK methods determines a sequence of approximation values $y_1, y_2, y_3 ...$ for the exact solution of the ODE system in a series of sequential time steps. In each of the time steps a fixed number s of

stage vectors are iteratively computed and combined to the next approximation vector in the following way:

for $l = 1, ..., s$ initialize stage vector $v_{(0)}^l$
  for $j = 1, ..., m$
    for $l = 1, ..., s$: compute new stage vector approximation $v_{(j)}^l$
compute new approximation vector $y_{k+1}^{(m)}$,
compute approximation vector $y_{k+1}^{(m-1)}$ for step size control.

The number m of iterations is given by the specific RK method. Each computation of a stage vector approximation requires an evaluation of the function f that describes the ODE to be solved. The advantage of the iterated RK methods for a parallel execution is that the iteration system of size $s \cdot n$ consists of s independent function evaluations that can be performed in parallel [8]. For systems of differential equations, an additional data parallelism can be exploited. Thus, the algorithm provides several possibilities for a parallel implementation. The computation of the stage vectors in one iteration j of the stage-vector computation can be performed on subsets of processors at the same time (*group implementation*) or alternatively by all processors one after another (*consecutive implementation*). The group implementation is a mixed task and data parallel implementation whereas the consecutive implementation is a pure data parallel realization.

4.2 Consecutive implementation

The computation time of the consecutive execution on p processors without the stepsize control can be modelled by the formula

$$T_{dp}(n,p) = \left(ms\left\lceil\frac{n}{p}\right\rceil + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \left(ms\left\lceil\frac{n}{p}\right\rceil + s\left\lceil\frac{n}{p}\right\rceil\right)T_f + n\,s\,t_{op}$$

where T_f denotes the time for the evaluation of *one* component of f and t_{op} denotes the time for the execution of one arithmetic operation. In each loop body each processor has to compute n/p components of the argument vector using $2s+1$ operations and n/p components of f. Since f and its access pattern are not known in advance, the complete argument vector has to be made available to each processor with a multi-broadcast operation. For the communication time, the following formula is used with t_{mb} from Table 1.

$$C_{dp}(n,p) = s\,m\,t_{mb}\left(p, \left\lceil\frac{n}{p}\right\rceil\right) + t_{mb}\left(p, \left\lceil\frac{n}{p}\right\rceil\right).$$

4.3 Group implementation

A group implementation of the iterated RK method uses s independent groups of processors where each group G_i of size g_i, $i = 1, \ldots, s$, is responsible for

forall $l \in \{1, \ldots, s\}$ do in parallel (*group parallelism*)
  forall processors $q \in G_l$ do (*data parallelism*) {
    compute $\lceil n/g_l \rceil$ components of $f(y_\kappa)$;
    initialize $\lceil n/g_l \rceil$ components of $\mu^1_{(0)}, \ldots, \mu^s_{(0)}$;
  }
for $j = 1, \ldots, m$ do (*sequential iteration*) {
  forall $l \in \{1, \ldots, s\}$ do in parallel (*group parallelism*)
    forall $q \in G_l$ do (*data parallelism inside groups*) {
      compute $\lceil n/g_l \rceil$ components of argument $\mu(l, j)$
      group-multi-broadcast of local components of $\mu(l, j)$;
      evaluate $\lceil n/g_l \rceil$ components of $1f(\mu(l, j))$;
      group-multi-broadcast of local components of $f(\mu(l, j))$
    }
forall processors q do (*data parallelism on all processors*)
  compute $\lceil n/p \rceil$ components of $y_{\kappa+1}$;
  multi-broadcast the computed components of $y_{\kappa+1}$;
}
stepsize control;

Figure 1. Group implementation for one time step of the iterated RK method.

computing the approximations of one specific stage vector. The pseudo-code in Figure 1 illustrates the structure of the program.

The processor groups should be of equal size, since the computation of each stage vector approximation requires an equal amount of computation. As it is possible that p is not a multiple of s, the group with the smallest number $g_{min} = \lfloor p/s \rfloor$ of processors determines the computation time

$$T_{gp}(n,p) = \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{p}\right\rceil\right)(2s+1)\,t_{op} + \left(m\left\lceil\frac{n}{g_{min}}\right\rceil + \left\lceil\frac{n}{g_{min}}\right\rceil\right)T_f + \left\lceil\frac{n}{g_{min}}\right\rceil s\,t_{op}.$$

The communication time is modelled by the runtime formula with the machine specific contention factor, see Equation (1), to reflect the concurrently executed multi-broadcast operations.

$$C_{gp}(n,p) = 2 \cdot m \cdot \tilde{t}_{mb}\left(g_{min}, \left\lceil\frac{n}{g_{min}}\right\rceil\right) + t_{mb}\left(p, \left\lceil\frac{n}{p}\right\rceil\right).$$

5. Runtime experiments

To validate the runtime formulas we have performed runtime tests on the and T3E and the CLiC for up to 128 processors. Since the execution time of the iterated RK method strongly depends on the ODE system to be solved, we consider two classes of ODE systems:

Sparse ODE systems. These ODE systems have a right-hand side function f for which the evaluation of each component has a fixed evaluation time that is independent of the size n of the ODE system (sparse function), i.e., the evaluation time of the entire function f consisting of n components increases linearly with the size of the ODE system.

Dense ODE systems. These ODE systems have a right-hand side function f for which the evaluation of each component has an evaluation time that increases linearly with n, i.e., the evaluation time of the entire function f increases quadratically with the size of the ODE system.

Dense ODE systems lead to the fact that the computation time usually dominates the communication time, i.e., the comparison of measured and predicted execution times shows how good the computation times are modelled. On the other hand, for sparse ODE systems the communication time plays an important role and the comparison also includes this part. We have used an iterated RK method with $s = 4$ stages that is based on an implicit RadauIIA method. This method leads to a convergence order of 7, if $m = 6$ iterations are executed in each time step. The runtimes shown in the following tables and figures are runtimes for one time step of the method and are obtained by averaging over a large number of time steps. The (measured and predicted) runtimes include the time for stepsize and error control.

On the T3E, the runtime of the *consecutive* implementations of the iterated RK method for *dense* ODE systems can be modelled very accurately and are not shown here. But sparse ODE systems also lead to good predictions. In this case, no concurrent message transmissions take place, and therefore the contention factor does not need to be used. For both cases, the predictions are quite accurate, but not as accurate as the predictions for dense ODE systems. Nevertheless, they are accurate enough to be used for predicting the effect of a task parallel implementation.

Figure 2 shows the deviations between measured and predicted runtime for a *group* implementation of the iterated RK method for *dense* (left) and for *sparse* (right) ODE systems on a T3E, again using the contention factor for the concurrent message transmissions because of a task parallel execution. Figure 3 illustrates the accuracy of the *group* implementation for *dense* (left) and *sparse* (right) ODE systems of large sizes on the T3E for different numbers of processors. In contrast to dense ODE systems, the solution of sparse ODE systems leads only to considerable speedups for processor numbers of up to

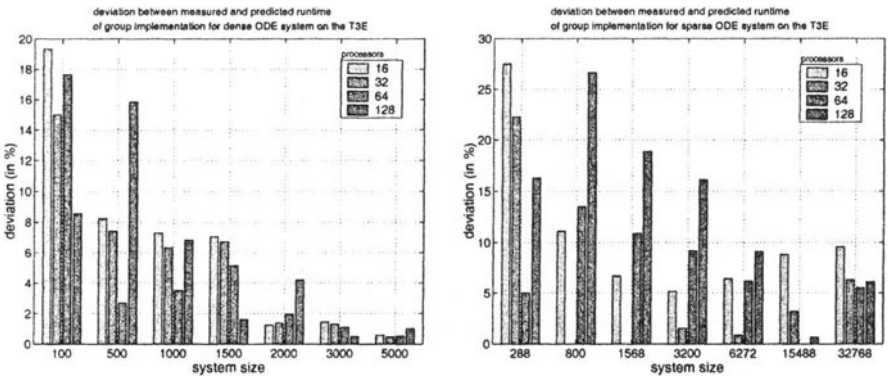


Figure 2. Deviation between measured and predicted runtime of group implementation for dense (left) and sparse (right) ODE system on the Cray T3E-1200.

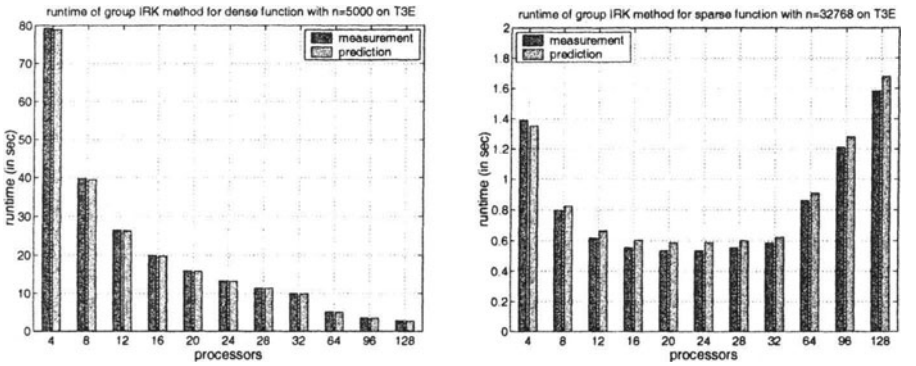


Figure 3. Measured and predicted execution times for a group implementation of one time step of the iterated RK method for a dense function of size $n = 5000$ (left) and for a Brusselator system of size $n = 32768$ (right) on a Cray T3E-1200.

16. For larger numbers of processors, the communication time dominates the computation times.

Figure 4 compares measured and predicted execution times for *group* implementations of the iterated RK methods for *dense* ODE systems on the CLiC. Figure 5 shows an illustration for a fixed message size and different numbers of processors. For this machine, the deviations between the measured and predicted execution times are much larger than for the T3E, especially for a larger number of processors. Although the deviations may be quite large, they can still be used as a rough estimate of the performance of a task parallel execution. The main reason for the large deviations are caused by the large increase of the communication time with an increasing number of processors. But also for a smaller number of processors, there are considerable deviations for large sys-

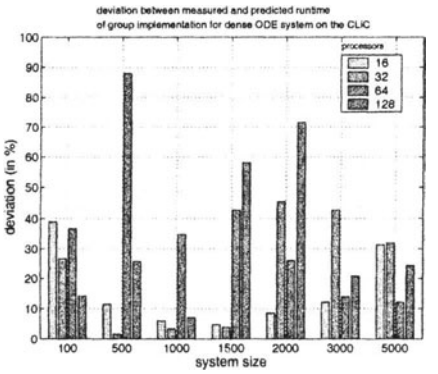


Figure 4. Deviation between measured and predicted runtime of a group implementation of one time step of the iterated RK method for a dense function on the CLiC Beowulf cluster.

tem sizes because of caching effects which are caused by the memory hierarchy of the single processors (Intel Pentium III). Such effects are much smaller on the Alpha 21164 processor of the T3E-1200 because of its different cache organization. The runtimes on the CLiC show that even for dense ODE systems, the machine only leads to satisfactory speedups for up to 32 processors. For larger processor numbers, the communication time and the network contention are too high.

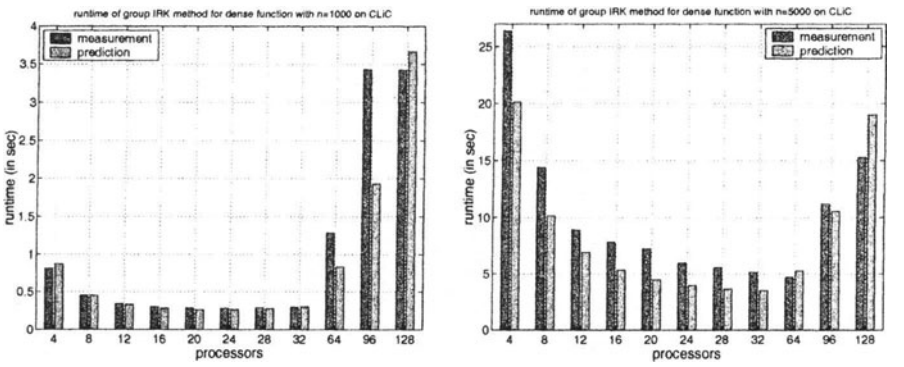


Figure 5. Measured and predicted execution times for a group implementation of one time step of the iterated RK method for dense ODE of size $n = 1000$ (left) and $n = 5000$ (right) on the CLiC Beowulf cluster.

6. Conclusions

In this article, we have shown that it is possible to model the execution times of mixed task and data parallel implementations by runtime formulas and that the use of a simple contention factor is sufficient to capture the interference of concurrent message transmissions. The runtime formulas model the execution times quite accurately for parallel machines like the T3E with a high-speed interconnection network. For a Beowulf cluster with an Ethernet-based network, the network contention caused by concurrent transmissions is much larger and it is more difficult to capture the effects by a simple contention factor. But the predictions are still reasonable and give a first impression of the possible effects of a task parallel realization.

References

[1] A. Alexandrov, M. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. Technical Report TRCS95-09, University of California at Santa Barbara, 1995.

[2] D.E. Culler, R. Karp, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. *4th Symp. on Principles and Practice of Parallel Programming*, 28(4):1–12, 1993.

[3] The MPI Forum. MPI: A Message Passing Interface Standard. Technical report, University Tennessee, April 1994.

[4] R. Foschia, T. Rauber, and G. Rünger. Modeling the Communication Behavior of the Intel Paragon. In *Proc. 5th Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, IEEE, pages 117–124, 1997.

[5] M. Hill, W. McColl, and D. Skillicorn. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[6] S. Johnsson. Performance Modeling of Distributed Memory Architecture. *Journal of Parallel and Distributed Computing*, 12:300–312, 1991.

[7] W.F. McColl. Universal Computing. In *Proceedings of the EuroPar'96*, Springer LNCS 1123, pages 25–36, 1996.

[8] T. Rauber and G. Rünger. Parallel Iterated Runge–Kutta Methods and Applications. *International Journal of Supercomputer Applications*, 10(1):62–90, 1996.

[9] T. Rauber and G. Rünger. PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. In *Proc. 11th Symp. on High Performance Computing Systems (HPCS'97)*, 1997.

[10] T. Rauber and G. Rünger. Modelling the runtime of scientific programs on parallel computers. In *Proc. ICPP-Workshop on High Performance Scientific and Engineering Computing with Applications (HPSFCA-00)*, pages 307–314, Toronto, Kanada, August 2000.

[11] Z. Xu and K. Hwang. Early Prediction of MPP Performance: SP2, T3D and Paragon Experiences. *Parallel Computing*, 22:917–942, 1996.

COLLECTIVE COMMUNICATION PATTERNS ON THE QUADRICS NETWORK*

Salvador Coll, José Duato, Francisco J. Mora
Department of Electronic Engineering
Technical University of Valencia, Valencia, Spain
scoll@eln.upv.es, jduato@gap.upv.es, fjmora@eln.upv.es

Fabrizio Petrini, Adolfy Hoisie
Performance and Architecture Laboratory, CCS-3
Los Alamos National Laboratory, Los Alamos, NM, USA
fabrizio@lanl.gov, hoisie@lanl.gov

Abstract The efficient implementation of collective communication is a key factor to provide good performance and scalability of communication patterns that involve global data movement and global control. Moreover, this is essential to enhance the fault-tolerance of a parallel computer. For instance, to check the status of the nodes, perform some distributed algorithm to balance the load, synchronize the local clocks, or do performance monitoring. Therefore, the support for multicast communications can improve the performance and resource utilization of a parallel computer. The Quadrics interconnect (QsNET), which is being used in some of the largest machines in the world, provides hardware support for multicast. The basic mechanism consists of the capability for a message to be sent to any set of contiguous nodes in the same time it takes to send a unicast message. The two main collective communication primitives provided by the network software are the barrier synchronization and the broadcast, which are both implemented in two different ways, either using the hardware support, when nodes are contiguous, or a balanced tree and unicast messaging, otherwise. In this paper some performance results are given for the above collective communication services, that show, on the one hand, the outstanding performance of the hardware-based primitives even in the presence of a high network background traffic; and, on the other hand, the limited performance achieved with the software-based implementation.

Keywords: interconnection networks, Quadrics, collective communication, multicast, performance evaluation

*The work was supported by the Spanish CICYT through contract TIC2000-1151-C07-05

1. Introduction

Current trends on high-speed interconnects include the availability of a communication processor in the network interface card [3, 11], which allows the implementation of high level messaging libraries without explicit intervention of the main CPU [4]; and the support for collective communications at hardware level [12], which outperforms traditional software-based multicast implementations. Both approaches can aid in the implementation of communication patterns which involve global data movement and global control.

Hardware support for multicast communication combined with the local processing power provided by network processors gives the opportunity of addressing several open problems in current and future medium- and large-scale parallel computers: scalability, responsiveness, programmability, performance, resource utilization and fault-tolerance. Many recent research results show that job scheduling and resource management techniques based on gang scheduling and coscheduling algorithms can provide solutions to these open problems [1, 7, 6, 10]. Another aspect where efficient multicast communication can play a key role is performance diagnosis and tunning, and performance control [9, 13]. By integrating dynamic performance instrumentation with configurable resource management algorithms and a real-time adaptive control mechanism, runtime systems could automatically configure resource managers. Such systems would increase achieved performance by adapting to temporally varying application behavior.

Hardware support for multicast communication requires many functionalities, that are dependent on the network topology, the routing algorithm and the flow control strategy. For example, in a wormhole network, switches must be capable of forwarding flits from one input channel to multiple output channels at the same time [14]. Unfortunately, these tree-based algorithms can suffer from blocking problems in the presence of congestion [15]. Also, the packets must be able to encode the set of destinations in an easy-to-decode, compact manner, in order to reduce the packet size and to guarantee fast routing times in the switches.

Software multicasts, based on unicast messages, are simpler to implement, do not require dedicated hardware and are not constrained by the network topology and routing algorithms, but they can be much slower than the hardware ones.

In previous work we analyzed in depth how hardware- and software-based multicasts are designed and implemented in the Quadrics network (QsNET) [12]. In this paper some modifications have been pertormed in the communication libraries to evaluate the underlying mechanisms that provide multicast support.

The initial part of the paper part introduces the mechanisms at the base of the hardware and software multicast primitives that, on their turn are at the base of

more sophisticated collective communication patterns as broadcasts, barriers, scatter, gather, reduce, etc.

In the second part we provide an extensive performance evaluation of two user-level collective communication patterns, barrier and broadcast, implemented using both hardware and software multicast algorithms.

The rest of this paper is organized as follows. Section 2 presents the basic mechanisms that support collective communication on the QsNET, while Section 3 gives a detailed description of the main collective communication services. Section 4 presents the experimental results and performance analysis. Finally, some concluding remarks and future directions are given in Section 5.

2. Collective Communication on the Quadrics Network

The QsNET is a butterfly bidirectional multistage interconnection network with 4 x 4 switches [11], which can be viewed as a quaternary fat-tree. It is based on two building blocks, a programmable network interface called Elan, and a low-latency high-bandwidth communication switch called Elite. It uses wormhole switching with two virtual channels per physical link, source-based routing and adaptive routing. Some of the most important aspects of this network are: the integration of the local memory into a distributed virtual shared memory, the support for zero-copy remote DMA transactions and the hardware support for collective communication [12].

The basic hardware mechanism that supports collective communication is provided by the Elite switches. The Elite switches can forward a packet to a set of physically contiguous output ports. Thus, a multicast packet can be sent to any group of adjacent nodes by using a single *hardware-based* multicast transaction. When the destination nodes are not contiguous a *software-based* implementation which uses a tree and point-to-point messages is used.

2.1 Hardware-Based Multicast

Hardware-based broadcasts are propagated into the network by sending a packet to the top of the tree and then forwarding the packet to more than one switch output as the packet is sent down the tree. Deadlocks might occur on the way down when multiple broadcasts are sent simultaneously [5]. This situation is avoided by sending broadcast packets always to a fixed top tree switch, thus serializing all broadcasts. In Figure 1 (a) it is shown that the top leftmost switch is chosen as the logical root for the collective communication, and every request, in the ascending phase, must pass through one of the dotted paths until it gets to the root switch. In Figure 1 (b) we can see how a multicast packet reaches the root node; the multiple branches are then propagated in parallel. If another collective communication is issued while the first one is still in progress, it is serialized in the root switch. The second multicast packet will be able to proceed

only after an EOP token cleans the circuit of the first communication (Figure 1 (c) and (d)). All nodes connected to the network are capable of receiving the multicast packet, as long as the multicast set is physically contiguous.

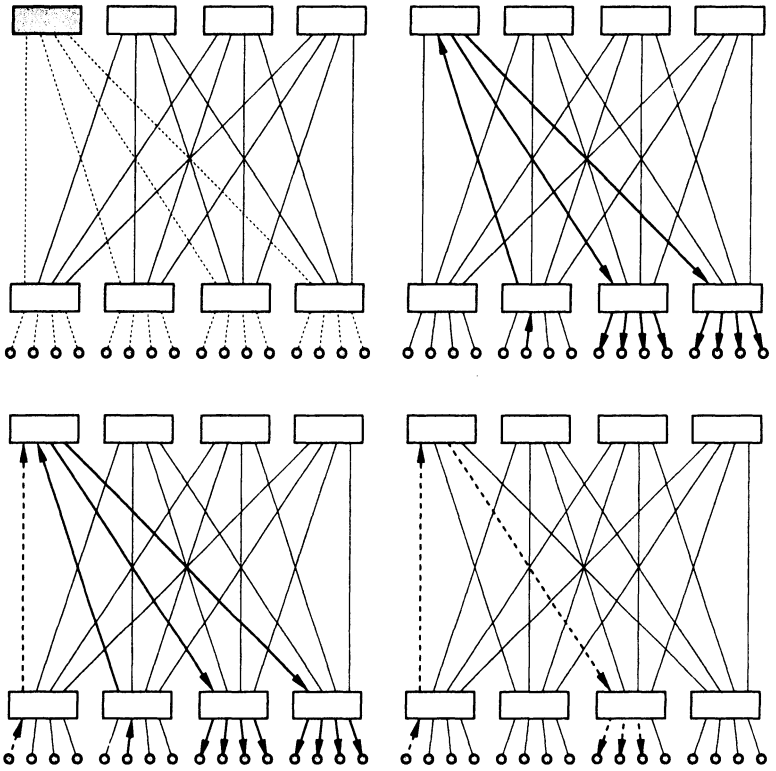


Figure 1. Hardware multicast

For a multicast packet to be successfully delivered, a positive acknowledgement must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgements, as pioneered by the NYU Ultracomputer [2], returning a single one to the source. Acknowledgements are combined in a way that the "worst" ack wins (a network error wins over an unsuccessful transaction, which on its turn wins over a successful one), returning a positive ack only when all the partners in the collective communication complete the distributed transaction with success.

2.2 Software-Based Multicast

The Quadrics communication libraries implement a software-based multicast algorithm to be used when the hardware support is not usable. The algorithm

uses a balanced tree to perform multicast transactions. Figure 2 shows the tree used for a 16-node group with the source at node 0. The source process sends a copy of the packet to its children, which, after receiving it, forward the packet to their children. Eventually all the processes will be reached. As it can be seen for the example on the figure, a broadcast would take 6 point-to-point transactions to be completed. This algorithm is performed by the thread processor included in the Elan. The thread processor can receive an incoming packet, do some basic processing and send one or more replies in few μs, without any interaction with the main processors.

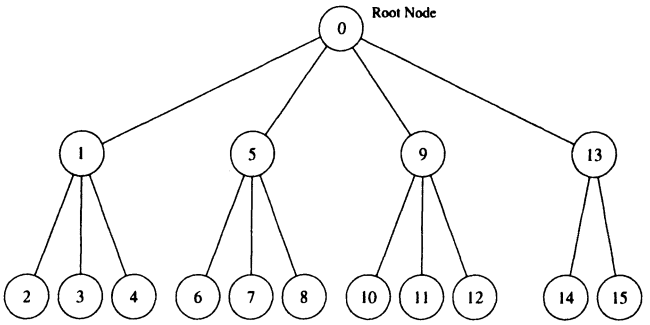


Figure 2. Balanced tree used by the software multicast algorithm

3. Barrier Synchronization and Broadcast

3.1 Barrier Synchronization

A synchronization barrier is a logical point in the control flow of a parallel program at which all processes in a group must arrive before any of the processes in the group are allowed to proceed. Typically, a barrier synchronization involves a logical reduce operation followed by a broadcast.

QsNET implements two different synchronization mechanisms in the communication libraries, a mixed software and hardware barrier with the name `elan_gsync()` and a purely hardware one called `elan_hgsync()`.

The algorithm implemented with `elan_gsync()` uses the balanced tree described in Section 2.2 to send the 'ready' signal to the process at the root node. Each process in the tree waits for 'ready' signals from its children, and when it receives all of them sends its own signal up to the parent process. When the root process receives all its 'ready' signals it performs a hardware broadcast which either sets an event (which all processes are waiting for) or writes a single word in a given memory location (which all processes are polling). If the destination nodes are not adjacent the same tree structure is used to distribute the data using point-to-point messages.

When a barrier operation is performed with one of the synchronization mechanisms (`elan_hgysnc()` or `elan_hgysncEvent()`), all processes in the group set a barrier sequence number in a system memory location (Figure 3(a)). All of them but the root node (which is the process with the lowest ID in the group) wait for a 'ready' signal (busy polling on a memory location with `elan_hgysnc()` or an event mechanism with `elan_hgysncEvent()`). The root process uses an Elan thread to send a special test-and-set broadcast packet (subfigure (b)). This packet spans all the processes and checks if the barrier sequence value in each process matches with its own sequence number (it does if the corresponding process reached the barrier). All the replies are then combined by the Elites on the way back to the root node which receives a single ACK token (subfigure (c)). If all the nodes are ready an EOP token is sent to the group to set an event or write a word to wake up the processes waiting in the barrier (subfigure (d)). It has to be noted that this mechanism is completely integrated into the network flow control.

3.2 Broadcast

The main communication primitive of the QsNET is the remote DMA. A DMA operation transfers data between local and remote address spaces (including Elan memory). In addition to providing point-to-point communication, DMAs can also be used to perform group-wide operations such as broadcast and flood DMAs (a flood is similar to a broadcast but the operation completes as soon as any of the destinations accepts the DMA). The effect of a write broadcast DMA is to copy the data from the source to the destination buffers of all the processes in the group. The implementation of the broadcast DMAs relies on all receiving processes having the destination buffer at the same virtual address, to obtain good performance.

Two different broadcast implementations are provided by the QsNET communication libraries: `elan_bcast()` and `elan_hbcast()`. Both must be called by all the processes in the group involved in the broadcast operation to guarantee that the receivers have allocated the buffers by the time the transaction is performed by the sender process. As a result, the broadcast is composed of two transactions: first, a barrier synchronization and, second, the broadcast itself. In both implementations, two types of memory resources can be used. On the one hand a global destination buffer, which has the same virtual address in all the processes (the communication library provides special memory allocation functions to do that), allows DMA transactions directly from one source to multiple destinations. On the other hand, if this memory allocation is not used, system buffers are utilized as intermediate copy space (this approach implies one copy at the source, and another copy at the destination).

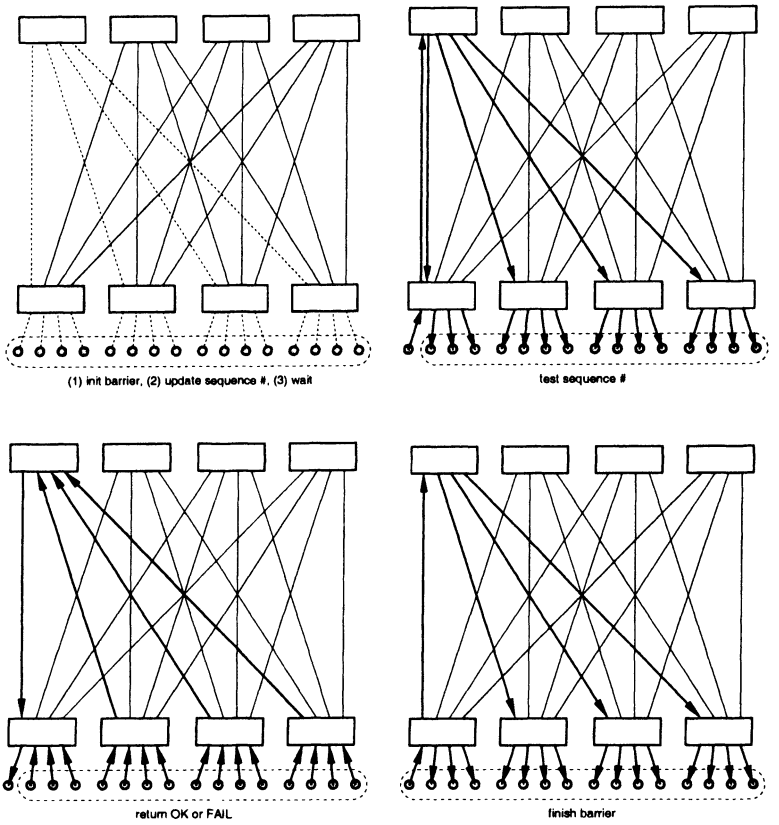


Figure 3. elan_hgsync() Barrier implementation

The `elan_bcast()` implementation uses a software-based synchronization for the first phase similar to that utilized by the first phase of `elan_gsync()` (Section 3.1). The second phase is triggered by an event set in the source node and is done using the hardware broadcast mechanism (if all the destination Elans are contiguous) or by means of a software-based broadcast (if the destination Elans are not). This transaction distributes the data and wakes up the processes waiting in the barrier performed during the first phase. This implementation provides better performance than a call to `elan_gsync()` (which involves a software-based synchronization and a broadcast) and a later broadcast to send the data.

The `elan_hbcast()` primitive calls `elan_bcast()` if the hardware broadcast mechanism is not available, for example when the nodes are not contiguous. If this mechanism is available, it performs a barrier to synchronize all the nodes

using `elan_hgsync()` (Section 3.1) and a hardware broadcast to distribute the data.

The Elan hardware broadcast can only write to the memory space of a single process per node since there is only a single context specified by the virtual process identifier. Hence, with multiple processes per node, the only way to use the hardware broadcast facility is to broadcast into an area of shared memory and then get the processes to copy from there. This has been optimized by using a FIFO like scheme that tries to overlap the broadcast with the copies.

4. Experimental Results

The performance of the collective communication services of the Quadrics network was evaluated on a 32-node cluster of Dell 1550, running Red Hat 7.1 Linux. Each node has two 1.13 GHz Pentium-III with 1GB of ECC RAM, and a Quadrics QM-400 Elan3 NIC attached to the network through a 66MHz/64-bit PCI bus.

4.1 Unidirectional Ping

To provide some basic performance results of the QsNET on our experimental testbed, we analyzed the latency and bandwidth of the network for unicast messages of different sizes sent between a pair of nodes. The communication buffers were placed either in main or in Elan memory. These tests provide a performance reference to consistently analyze the results on collective communication.

The bandwidth of the unidirectional ping is shown on Figure 4 a). The asymptotic data bandwidth is 336 MB/s and is obtained when the buffers are placed in the Elan memory. Taking into account the data payload and the overhead introduced by the message header (routing tags, CRC, etc.), the delivered peak bandwidth is 396 MB/s, or 99% of the nominal bandwidth (400 MB/s). With buffers in main memory the peak bandwidth is 324 MB/s. These results also show that the PCI interface running at 66 MHz provides a good performance.

Figure 4 b) shows the latency for messages shorter than 4 KB. With Elan memory buffers the latency is almost constant at 2.1 μs for messages up to 64 bytes, because these messages can be sent using a single transaction. We note a slight increase in latency with main memory buffers of 0.3 μs for messages up to 16 bytes and of 1 μs for messages up to 4 KB.

4.2 Collective Communications

The barrier synchronization and broadcast primitives provided by the QsNET have been tested using configurations ranging from 4 to 32 nodes. Results have been obtained by averaging the metrics over 10000 consecutive tests. Average latency results are reported for the barrier synchronization tests. For the broad-

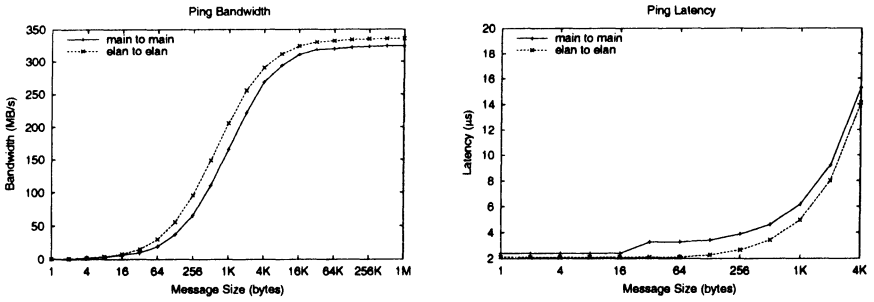


Figure 4. Unidirectional Ping

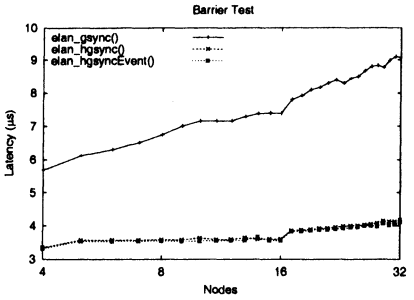


Figure 5. Barrier Synchronization

cast tests bandwidth and latency are reported. Results for larger configurations can be found in [12]. A scalability analysis for broadcast is presented in [8].

In addition, tests with background traffic have been performed to analyze the behavior of the collective communications under network contention. This background traffic is generated by 32 processes running in 32 nodes, with all nodes injecting messages into the network at maximum load. The goal of these tests is to identify the performance degradation experienced by the collective communication in the presence of congestion. The traffic pattern used to generate background traffic has been complement: the node with binary coordinates $a_{n-1}, a_{n-2}, \ldots, a_1, a_0$ communicates with the node $\overline{a_{n-1}, a_{n-2}, \ldots, a_1, a_0}$. This pattern was selected because it uses all the network links at the same time.

To guarantee that the performance degradation of the collective communication is only due to the network contention and not to scheduling issues, the background traffic generation and the collective communication benchmark were run in distinct processors.

4.2.1 Barrier Synchronization. Figure 5 shows the average time required to perform a barrier synchronization in an empty network. Results for

the three primitives provided by the libraries (Section 3.1) are shown versus the number of nodes. We can see that the hardware-based implementations of the barrier (`elan_hgsync()` and `elan_hgsyncEvent()`) provide the best results when compared to the software-based implementation (`elan_gsync()`), both in absolute performance and in scalability. The latency of the software-based implementation grows as the logarithm of the number of nodes (approximately $1\mu s$ each time the number of nodes is doubled). In this case the average latency to synchronize 32 nodes is $9.1\mu s$. On the other hand, the hardware barriers (which show negligible differences between them) provide an average latency of $4.2\mu s$ for 32 nodes.

The behavior of the barrier synchronization has been analyzed by performing tests with complement background traffic. The results depicted in Figure 6 show that the buffer allocation of the background traffic has no significant effect. The network is the bottleneck in this case, not the PCI bus. The software barrier is significantly affected by the background traffic, the slowdown is 60 in the worst case of 32 nodes. On the other hand, there is little impact on the hardware barriers, whose latency is only doubled (1.8 slowdown).

The scalability is also affected by the background traffic. The latency of the hardware-based implementations with the number of nodes increases 24% (when the number of nodes varies from 4 to 32) with no background traffic, and 43% with background traffic. On the other hand, the latency increase of the software-based synchronization is 60% and 180% respectively. The software-based barrier latency scalability is shown to be more sensitive to complement background traffic than the hardware-based barriers.

These results show an important outcome, from a practical point of view, the hardware-based implementation of the barrier can be considered insensitive to network contention.

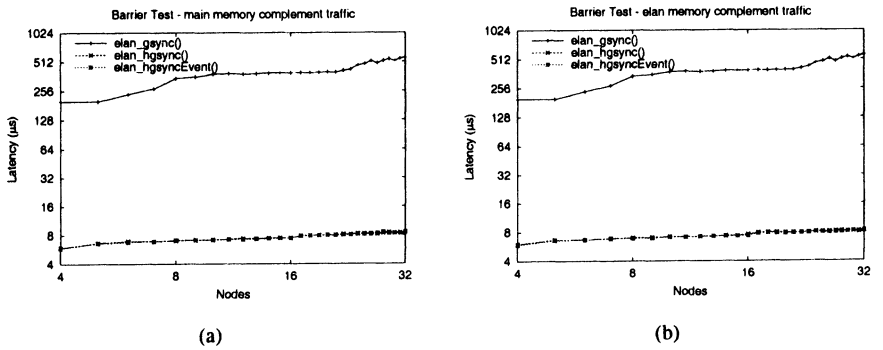
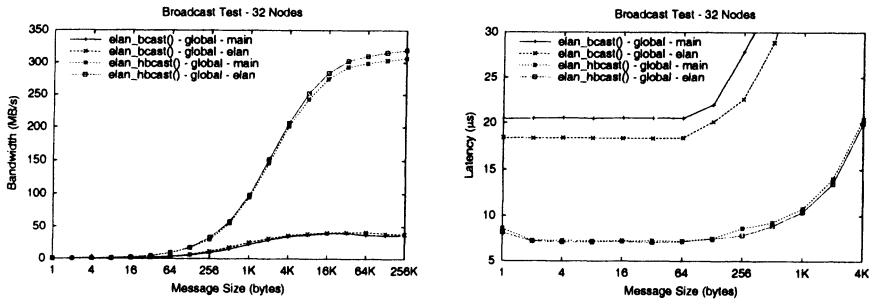


Figure 6. elan_hgsync() Latency with Contention

4.2.2 Broadcast. As mentioned in Section 3.2, the `elan_bcast()` primitive by default uses a hardware multicast when all the destination nodes are contiguous. In order to perform a fair comparison between the hardware- and the software-based multicast mechanisms, the original implementation of `elan_bcast()` has been modified to use the tree-based algorithm, even in the case where the destination nodes are adjacent. As described in Section 3.2 the software-based broadcast primitive implementation uses the hardware mechanism if available. Thus, the results reported in this section refer to the modified version of `elan_bcast()`.

Figure 7 shows the results obtained with broadcast over 32 nodes using both algorithms supported by the system libraries (Section 3.2) with buffers globally allocated in main and Elan memory, that is, with the same virtual address in all processes. As expected, the best performance is obtained with the hardware-based broadcast with buffers in Elan memory. In this case the measured bandwidth for 256 KB messages is 319 MB/s, which is 95% of the unicast bandwidth (Section 4.1). With buffers in main memory the peak bandwidth is 306 MB/s, or 95% of the unicast bandwidth. The asymptotic bandwidth of the software-based implementation is 40 MB/s (8 sending steps to reach the last node in a 32-node network), which is worse than what it would be expected with a binary-tree implementation (5 sending steps to reach 32 nodes). For all the implementations the latency for messages shorter than 64 bytes is constant since those messages are sent using a single transaction. The hardware-based broadcast latency is lower than 8 μs for messages up to 256 bytes, with no significant effect of the memory allocation, while the software-based broadcast takes 10 μs longer when using Elan memory buffers and 12 μs longer with main memory buffers.

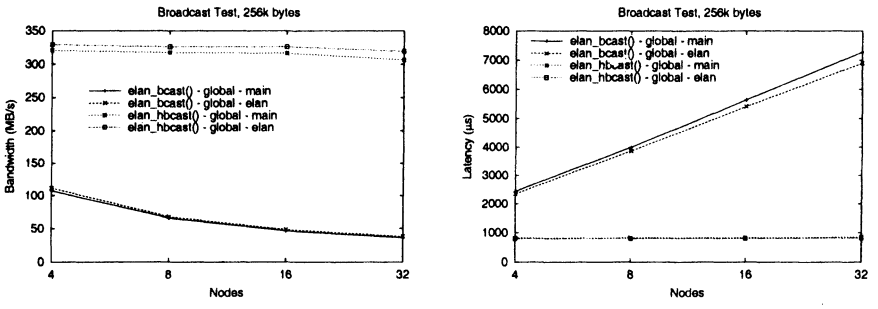


(a) (b)

Figure 7. Broadcast

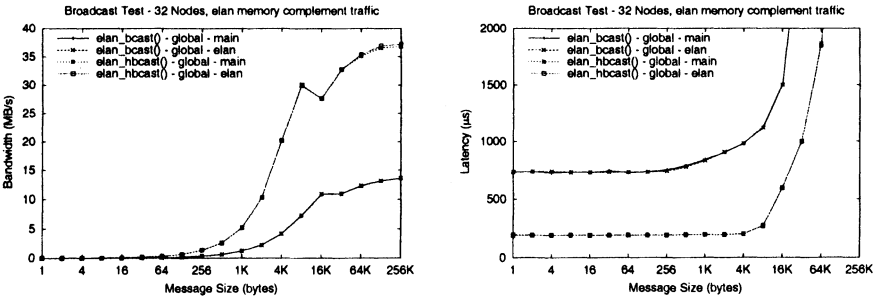
Bandwidth and latency versus the number of nodes for 256 KB messages are depicted in Figure 8. Regarding the hardware-based broadcast, both per-

formance metrics are almost insensitive to the number of nodes (for the tested configurations), slowdowns between 3 and 4% were measured. On the other hand, when the software-based broadcast is used, a significant performance degradation occurs when the number of nodes increases due to the logarithmic behavior of the tree-based implementation. In this case the slowdown is 66% when the number of nodes increases from 4 to 32.



(a) (b)

Figure 8. Broadcast Scalability



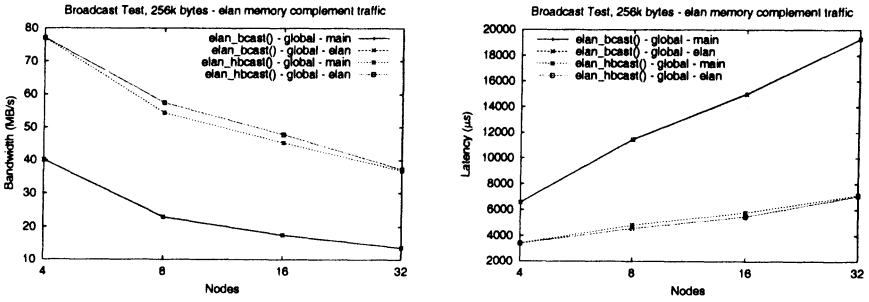
(a) (b)

Figure 9. Broadcast with Contention

In the presence of network contention (Figure 9) the broadcast performance decreases significantly. Similar results are obtained either with background traffic using main or Elan memory (due to space limitations only results for background traffic in Elan memory are shown). The asymptotic bandwidth for the hardware-based broadcast is 37 MB/s and the software-based broadcast gets 14 MB/s. The hardware-based implementation suffers from a higher degrada-

tion in the presence of background traffic since there is higher contention with the background traffic to perform the link reservation.

In terms of scalability (Figure 10) the four alternatives suffer from the same performance degradation as the number of nodes increases. Although his effect slows down as the number of nodes is increased, we need to further investigate this behavior by analyzing larger configurations.



(a) (b)

Figure 10. Broadcast Scalability with Contention

5. Conclusion

In this paper, we present a description and evaluation of the Quadrics interconnection network (QsNET) support for collective communication.

The underlying mechanism that provides the hardware support for collective communication is presented. After that, the two basic communication patterns provided by the system software, barrier synchronization and broadcast, are described.

An experimental evaluation of hardware-based and software-based implementations of these services has been performed on a 32-node cluster. Our experiments show that the time to complete a hardware-based barrier synchronization on the whole set of nodes is as low as 4.2 μs, with very good scalability for the network configurations tested. In the presence of network contention, the average latency for the hardware barrier is 8.5 μs, with 95% of the synchronizations taking less than 16 μs. From a practical point of view the hardware-based barrier can be considered insensitive to network contention.

Good latency and scalability are also achieved with the software-based synchronization, which completes in 9.1 μs on an empty network. On the other hand, it is shown to suffer a significant performance degradation with background traffic.

Regarding the broadcast, the hardware-based implementation can deliver a sustained bandwidth of 319 MB/s (95% of the point-to-point bandwidth) with less than 8 μs latency for messages up to 256 bytes when using Elan memory buffers (306 MB/s, 8 μs with main memory buffers). The software-based broadcast delivers an asymptotic bandwidth of 40 MB/s for any memory allocation (the bottleneck in this case is the algorithm itself, not the PCI bus).

Contention tests, done in the presence of high network load, show that the broadcast maintains reasonably good performance (i.e. less than 200 μs to deliver messages up to 4 KB). In this case the hardware-based broadcast outperforms the software-based broadcast thanks to its hardware-based synchronization and packet transmission mechanism.

Overall, our analysis shows the potential of the interconnect to efficiently support large-scale collective communication, even in the presence of high network contention. On the other hand, while the hardware support is shown to make possible extremely good performance, its usage is limited to those cases where all the destination nodes are physically contiguous. Otherwise, for example with a single faulty node, a tree-based software implementation is used. This situation is likely to happen in current an future high-performance parallel machines, soon to reach tens of thousand of processors. This fact makes the hardware support unusable in practice. As future work, we plan to address this problem to overcome the impact of a few faulty nodes (or components) on the performance of collective communications. This will make a great impact not only on applications performance but on the behavior of the whole system. In particular if modern resource managers with job launching and process scheduling functions, which rely on efficient collective communication, are used.

Acknowledgments

The authors would like to thank the Quadrics team, David Addison, Jon Beecroft, Robin Crook, Moray McLaren, David Hewson, Duncan Roweth and John Taylor, for their invaluable support.

References

[1] Andrea Carol Arpaci-Dusseau. Implicit coscheduling: coordinated scheduling with implicit information in distributed systems. *ACM Transactions on Computer Systems*, 19(3):283–331, 2001.

[2] G. Bell. Ultracomputer: a Teraflop before its time. *Communications of the ACM*, 35(8):27–47, 1992.

[3] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[4] Darius Buntinas, Dhabaleswar Panda, P. Sadayappan. Performance Benefits of NIC-Based Barrier on Myrinet/GM. In *Workshop on Communication Architecture for Clusters (CAC '01)*, San Francisco, CA, April 2001.

[5] José Duato, Sudhakar Yalamanchili, Lionel Ni. *Interconnection Networks: an Engineering Approach*. IEEE Computer Society Press, 1997.

[6] Fabrizio Petrini, Wu-chun Feng. Buffered Coscheduling: A New Methodology for Multitasking Parallel Jobs on Distributed Systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2000, IPDPS2000*, Cancun, MX, May 2000.

[7] Dror G. Feitelson, Morris A. Jette. Improved Utilization and Responsiveness with Gang Scheduling. In Dror G. Feitelson and Larry Rudolph (Eds.), *Job Scheduling Strategies for Parallel Processing*, volume 1291 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[8] Eitan Frachtenberg, Fabrizio Petrini, Juan Fernandez, Scott Pakin, Salvador Coll. Storm: Lightning-fast resource management. In *IEEE/ACM SC2001*, Baltimore, MD, November 2002.

[9] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, Karen L. Karavanic R. Bruce Irvin, Krishna Kunchithapadam, Tia Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, November 1995.

[10] Fabrizio Petrini. Scaling to Thousands of Processors with Buffered Coscheduling. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002.

[11] Fabrizio Petrini, Wu chun Feng, Adolfy Hoisie, Salvador Coll, Eitan Frachtenberg. The Quadrics Network: High Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, January-February 2002.

[12] Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg, Adolfy Hoisie. Hardware- and Software-Based Collective Communication on the Quadrics Network. In *IEEE International Symposium on Network Computing and Applications 2001 (NCA 2001)*, Boston, MA, October 2001.

[13] Randy L. Ribler, Jeffrey S. Vetter, Huseyin Simitci, Daniel A. Reed. Autopilot: Adaptive Control of Distributed Applications. In *7th IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[14] Rajeev Sivaram, Dhabaleswar Panda, Craig Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, New Orleans, LA, October 1996.

[15] Rajeev Sivaram, Dhabaleswar Panda, Craig Stunkel. Multicasting in Irregular Networks with Cut-Through Switches using Tree-Based Multidestination Worms. In *Parallel Computing, Routing, and Communication Workshop, PCRCW'97*, Atlanta, GA, June 1997.

II

PERFORMANCE TOOLS AND SYSTEMS

THE DESIGN OF A PERFORMANCE STEERING SYSTEM FOR COMPONENT-BASED GRID APPLICATIONS

Ken Mayes, Graham D. Riley, Rupert W. Ford, Mikel Luján, Len Freeman
Centre for Novel Computing, Department of Computer Science
University of Manchester, Manchester, UK
{ken,griley,rupert,mlujan,lfreeman}@cs.man.ac.uk

Cliff Addison
Computing Services Department
University of Liverpool, Liverpool, UK
caddison@liv.ac.uk

Abstract A major method of constructing applications to run on a computational Grid is to assemble them from components - separately deployable units of computation of well-defined functionality. Performance steering is an adaptive process involving run-time adjustment of factors affecting the performance of an application. This paper presents a design for a system capable of steering, towards a minimum run-time, the performance of a component-based application executing in a distributed fashion on a computational Grid. The proposed performance steering system controls the performance of single applications, and the basic design seeks to separate application-level and component-level concerns. The existence of a middleware resource scheduler external to the performance steering system is assumed, and potential problems are discussed. A possible model of operation is given in terms of application and component execution phases. The need for performance prediction capability, and for repositories of application-specific and component-specific performance information, is discussed. An initial implementation is briefly described.

Keywords: performance steering, component-based application, Grid computing, Globus

1. Introduction

The aim of this paper is to present a design for a system capable of controlling, or steering, the performance of a component-based application.

A component is a unit of computation which has a well-defined functionality, and which can be composed with other components to create an application. Since a component can be individually deployed on to a hardware platform, component-based applications are, in the general case, distributed applications. Component-based applications are good candidates for deployment on to computational Grids, and several component frameworks exist to support the interaction of components e.g. [9]. It is in this context that performance steering of components and applications will be considered.

Two kinds of computational steering have been identified: *application steering* and *performance steering* [23]. Application steering "lets researchers investigate phenomena during the execution of their application...". On the other hand, performance steering "seeks to improve an application's performance by changing application and resource parameters during runtime". Whereas application steering tends to be interactive, performance steering can be quantified and automated, allowing a long-running application to adapt to a changing execution environment. It would be equally valid to apply the term "performance control" to such an autonomous adaptive system.

The term "performance steering" is defined here as being the adaptive process of adjusting, at run-time, factors affecting the performance of an application so that the application execution time is minimised.

The performance steering system discussed here is being developed within the RealityGrid project [16]. The aim of this project is to allow scientists to explore component-based simulations of physical phenomena. However, the project includes an investigation into autonomous performance steering of the simulations.

This paper discusses possible execution environments for, and the nature of, component-based Grid applications, in order to motivate the performance steering design. The entities incorporated in the design are also discussed. Possible interactions with general Grid framework software are presented. An initial implementation of the basic design is briefly described, and issues to be investigated by incremental implementation are given.

2. Execution environment of component-based applications

In order to design a performance steering system for a component-based application, it is necessary to examine the nature of such computations, and the features of the likely execution environment.

A schematic view of a simple component-based application executing on a computational Grid is shown in Figure 1. A computational Grid is a network of heterogeneous machines which can inter-operate to execute applications. The essence of a computational Grid is this inter-operation, supported by so-called *grid middleware* – a suite of more-or-less integrated software which interacts to provide security, resource management and information services. Although it is possible to implement component-based applications directly on this Grid middleware layer, it is also possible to have a further middleware layer to support components specifically. Such *component frameworks* allow greater flexibility and generality in the interactions between components.

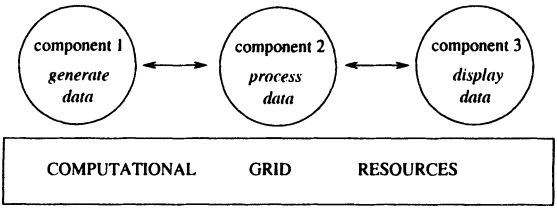


Figure 1. An simple example of a component-based application.

Central to such middleware layers is some *resource scheduler*, responsible for allocating Grid resources to applications submitted for execution. The GrADS project presents a Grid application execution framework which includes a "scheduler/resource negotiator"[12]. This scheduler is "responsible for choosing Grid resources appropriate for a particular problem run based on that run's characteristics...". Similarly component frameworks, such as that being produced by the ICENI project, will include a scheduler which "can schedule according to the available resources and requirements" [9].

Such Grid resource schedulers are referred to here as *external resource schedulers* because they are external to the performance steering system.

An application, consisting of multiple components to be executed on a computational Grid, requires that multiple resources are allocated to it simultaneously [5]. There are two factors associated with resource scheduling for applications consisting of components which are of relevance here. The first is the *allocation* of a set of resources to the application as a whole. The second is the *distribution* of the components of the application over these resources.

Few assumptions can be made about the nature of this external resource scheduler. At its simplest it may be in the form of a Globus Resource Specification Language script written by the application owner [11]. Alternatively, the scheduler could be one of the more sophisticated resource schedulers/negotiators. In the latter case, there must be some specification of the resource requirements of the application when it is submitted to the external resource scheduler. However, whilst considering the sophistication of possible

Grid resource schedulers, it is perhaps worth noting that "Currently, the most common current grid scheduler is the user" [19].

2.1 Resource allocation within a steered application

Ideally, it would be assumed that the allocation and possibly initial distribution of resources is carried out by the external resource scheduler, and that the run-time re-distribution of resources to components is the responsibility of the performance steering system.

That is, a steered application would have a fixed set of resources allocated to it. The partitioning of those resources between components of the application is under control of the performance steering system. Re-assignment of these given resources would enable the steering system to adjust the performance of its components. However, there may be specific hardware requirements associated with particular components (for example, a visualisation component) which would have to be taken into account.

3. Performance steering of component-based applications

It is reasonable to assume that performance-steered applications will be executing in the presence of other applications running on the same computational Grid. However, the performance steering system cannot influence these other applications. That is, a steered application is steered in isolation. From the point of view of the performance steering system and its single steered application, all other applications are invisible.

It is assumed that interactions between the resource requirements of competing applications, whether steered or not, will be dealt with by the external resource scheduler. Having an application-oriented approach to control performance efficiency for a single application is not new. The AppLeS scheduler emphasised application-specific information in meeting performance criteria [2]. Similarly, in the GrADS framework the performance of each application is managed by its own "application manager"[12].

3.1 Application consists only of components

A steered application consists only of interacting components. That is, it is assumed that there is no execution of application code external to the components. There is nothing that, for example, actively coordinates the behaviour of the components. This means that the performance of the application is determined solely by its components and their interactions. These interactions occur via the "middleware" which joins together the components, including the performance steering system. The performance steering system will not steer the performance of the middleware.

3.2 Component performance

An application is essentially a flow of control and of data through a set of interacting components. In general a component itself may be complex, consisting of several execution phases, where each phase can have different performance characterists or different resource requirements. Additionally, it may be possible to have alternative implementations of the same component, each using a different algorithm or data structure.

There is, in the general case, no restriction on the interactions between components. That is, the performance of some set of components may limit the performance of the whole application. Since the performance characteristics of a component may change over its execution, different components may, at different points, become rate-limiting. That is, the critical path through the application may change.

In order that a component can be steered, the component must provide some mechanisms or *effectors* which can be used by the steering system to modify the behaviour of the component.

3.2.1 Execution phases. The execution of the application, and its constituent components, is viewed as progressing through a series of phases. Each phase may have different characteristics from other phases. The usefulness of considering execution phases is that they enable performance to be characterised and measured at well-defined intermediate stages during run-time. Similarly, there is the possibility, between phases, of re-distributing resources to components.

4. Application performance QoS and steering

In an environment like a computational Grid, where an application may be running as processes on a diverse set of platforms, there is a need for Quality of Service (QoS) aware resource-sharing which prevents greedy applications from consuming resources shared with other applications [13]. The provision of a QoS-aware resource scheduler can be based on predicting the application behaviour or resource usage, with subsequent run-time monitoring to detect failure to achieve the expected performance or usage.

The main point here is that, in the case where there does exist some QoS performance contract for a Grid application, it is assumed that the external resource scheduler will be responsible for meeting QoS-related performance targets. This means that the role of the performance steering system is simply to optimise application performance with available resources. The sole criterion to be used for this optimisation will be execution time.

It may happen that during run-time a QoS-aware external resource scheduler alters the allocation of resources and/or the distribution of components over

those resources. That is, both external resource scheduler and performance steerer would be re-distributing resources between components. This situation would require some coupling between scheduler and steerer in order to avoid chaos.

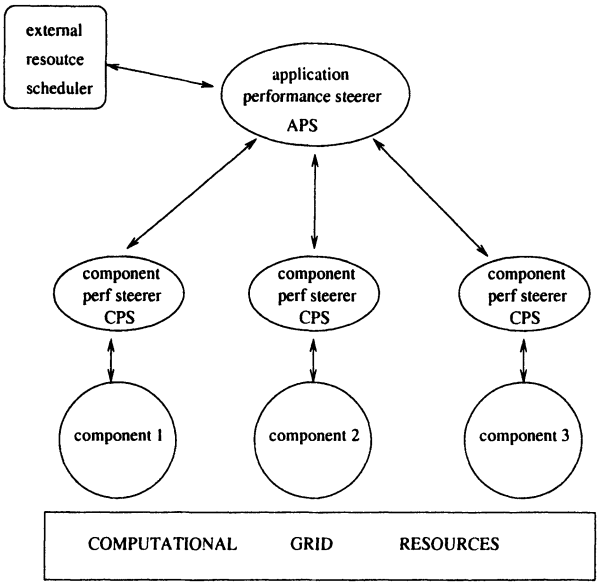


Figure 2. The structure of an application consisting of three interacting components. Steering is provided at two levels: at the application level and at the component level. The application-level steerer (APS) controls the entire application via the component-level steerers (CPS). The Grid middleware is assumed to provide some external resource scheduler which is responsible for allocating resources to the application.

5. Design of a performance steering system

Having introduced issues about the execution environment and application structure, it is possible to describe the basic design of a performance steering system. As was shown in Figure 1, the components are distributed across the resources of a computational Grid. The entity which creates this initial distribution is some middleware external resource scheduler. In the general case, each component may be executing on a different platform from the others. Thus, in order to achieve maximum performance of each component, there should be some component-local performance steerer which has access to component-specific performance-related information and effectors. However, there needs also to be some entity which monitors the progress of the application as a whole, and which can, for example, redistribute allocated resources to components which have become rate-limiting.

Thus the basic design of the performance steering system is to divide functionality into two levels: the application level and the component level. That is, there is an *Application Performance Steerer* (APS) which is responsible for steering or controlling the performance of the application as a whole. At the component-level, each component has an associated *Component Performance Steerer* (CPS) which is responsible for steering or controlling the performance of only that component (Figure 2).

The APS is responsible for application-wide activity, such as re-distributing resources to components in order to affect performance. Similarly, any necessary interactions between the external resource scheduler and the performance steering system should occur via the APS. On the other hand, it is also assumed that there are component-specific behaviours that can be handled by the CPS, such as changing component performance by changing the algorithm used by a component.

It is implicit in the roles of APS and CPS, that the steering system should have access to some *performance prediction capability* and to some *repository of performance information.* A more comprehensive design, showing these facilities added to the basic design, is given in Figure 3. Additional features shown here include a resource usage monitor and a component loader responsible for starting, restoring or migrating the component. In more concrete terms, this loader might be some "container" as in Enterprise Java Beans or ICENI [6] [10], or be implemented by some Unix server.

6. Basic roles of APS and CPS

Within the context of this design, the APS and CPS have distinct roles. Both APS and CPS roles require some performance prediction facility.

6.1 Role of the APS

The role of the APS can be simply expressed:

To distribute available resources amongst components such that the predicted performance of components gives a minimum predicted application execution time.

That is, the APS is given a set of resources by the external resource scheduler. The APS must attempt to distribute these resources between the components in an optimal manner, so that the application execution time will be minimised. Finding this optimal distribution involves:

1 predicting the performance of each component for a given resource distribution;

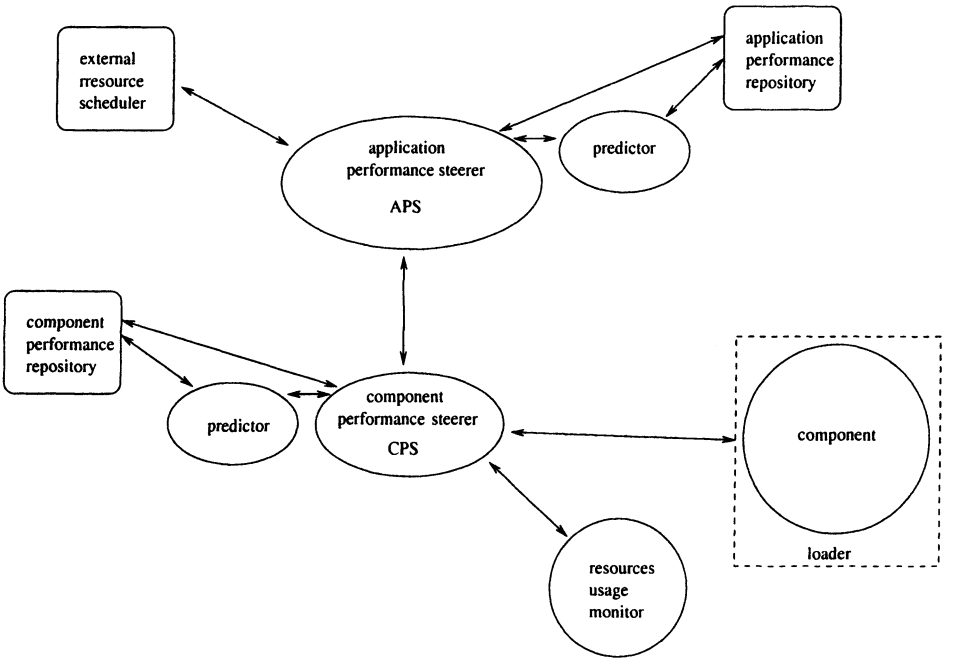


Figure 3. Entities in the performance steering system design. For simplicity, only one component is shown.

2 predicting the performance of the application for each set of predicted component performances.

Thus performance prediction activity is central to the APS role. Performance-related information will be available via some performance repository.

6.2 Role of a CPS

The role of a CPS is simply expressed:

To utilise resources and component performance effectors so that the predicted execution time of the steered component is minimised.

The APS allocates a set of resources to each component. Each CPS then steers its component so as to achieve minimum execution time using those resources. A CPS is assumed to have access to a repository of performance-related information for that component. In addition, the CPS may take into account resource-usage information available for the platform on which the component is executing.

7. Performance steerer model of operation

7.1 APS model of operation

One possible view of the operation of the APS can be given in terms of a cycle of activities, shown in Figure 4. An assumption made about an application to be performance-steered is that its execution proceeds in phases (see section 3.2.1), and that there are *progress points* between the phases. APS activity is assumed to occur at these progress points. This APS activity is largely a decision-making process which may result in a new resource distribution. This *simplified model* allows activities to be considered more cleanly. That is, in this model all the APS activity occurs with synchronised arrival of all inputs to the APS decision-making process, and synchronised re-distribution of resources to components. In practice this synchronisation constraint will be relaxed. The synchronisation methods used will be determined on the basis of costs and benefits of possible mechanisms. However, the basic APS activities should be unchanged and can be described within the framework of the simple model presented.

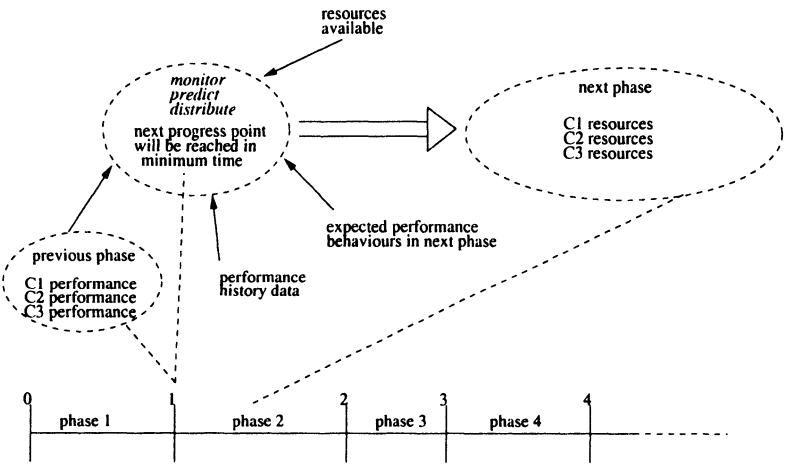


Figure 4. A model of how the application performance steerer operates in the context of an application consisting of three components (C1, C2 and C3). Execution proceeds in phases separated by progress points. At each progress point the application performance steerer determines the best distribution of resources between components for the next phase. Arrows show possible inputs to this process.

At each progress point, the APS generates a resource distribution for the components. The APS allocates resources such that the predicted performances of components will minimise application execution time. This implies that, at progress points, each component must be able to be restructured - the component must be at some "safepoint" in its execution. The CPSs then steer the performance of components through the next phase.

7.2 CPS model of operation

As with the application, each component can also be viewed as consisting of phases divided by progress points (for example, the timesteps in which the component may execute). However, there may be many component progress points within a one phase of the application (Figure 5).

In this model, a CPS can steer its component autonomously between application progress points. In Figure 5, the CPS can steer its component during application phase 1, and to a lesser extent in phases 3 and 4. During phase 2, there are no component progress points, so the CPS has very little opportunity to adjust to any resource changes which may be imposed by the APS. The greater the number of component progress points in an application phase, the greater is the opportunity for the CPS to steer its component during that phase. Greater autonomy for the CPS means less global activity, and opportunities for greater efficiency.

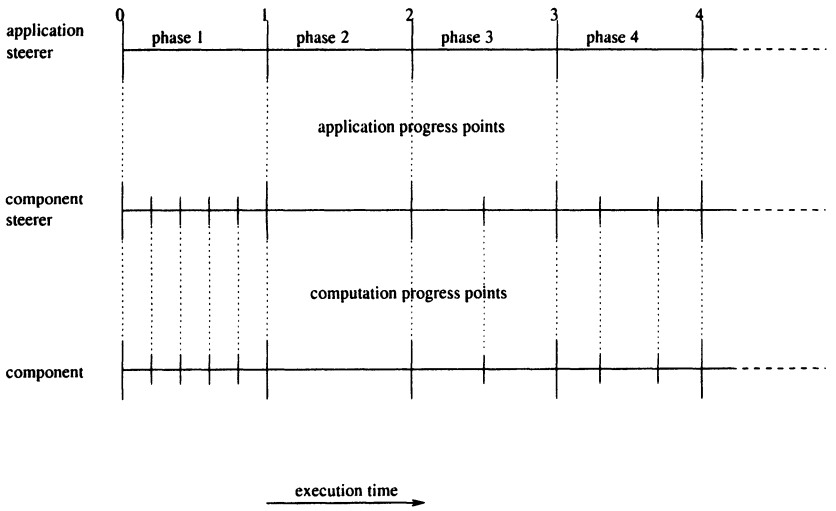


Figure 5. A model of the relationship between computation, computation performance steerer and application performance steerer. Information and control are exchanged at progress points indicated by the dotted vertical lines. There can be any number of computation progress points in one application phase. Application phases may be of different durations.

8. Performance steering support facilities

The design presented in Figure 3 includes performance repository, performance predictor and resource usage monitor facilities. These facilities are discussed in the following three subsections.

8.1 Performance information repository

The proposed system will steer application performance by run-time adjustment of performance-related factors. These adjustments will be carried out on the basis of their expected effect on performance. That is, the performance steering system must be capable of making some sort of performance prediction[1].

The performance steering system will thus require information about the application in order to support performance prediction for the entire application. Similarly, component-specific performance-related data should also be available to the system. Performance steering is an adaptive, run-time, process, and so information must be available for different algorithms, phases of execution, and resource configurations. This performance-related data will need to be available at run-time from some "repository". This repository must allow the addition and retrieval of a variety of performance history data, at run-time.

The Global Grid Forum Grid Database Access and Integration Services Working Group are developing standards for database services on the Grid [1]. It may be that this work can be used in designing the performance data repository. However, performance-related information has unique characteristics: it is usually short-lived, is updated more often than it is queried, and is often stochastic [22].

Performance-related information includes:

1 information about the application and components, such as complexity of available algorithms, expected data and control flow between components, component-specific hardware requirements.

2 models to compute expected execution behaviour for different platform configurations.

3 dynamic data about the previous performance history of the application: previously measured work-flows through its components, relation between performance and number of processors, and so on.

That is, performance-related data will need to be maintained about the performance of the application as a whole, and about the performance of each component. More specifically, it is apparent that there must be a division of the repository between dynamic short-lived data, and static long-lived data. The static portion of the repository should be optimised for queries whereas the dynamic portion should be optimised for updates. As an efficiency concern,

[1] The performance steering system is similar to a QoS-aware external resource scheduler in that it must co-schedule available resources to executables on the basis of some set of performance expectations.

component information should be maintained local to the component. The processing of dynamic history data into static data can occur "off-line" between runs, or during run-time. The repositories will be closely associated with the performance predictors of both APS and CPS. In addition, the APS may need a copy of data about components.

8.2 Performance prediction

Both the APS and CPS must have access to performance prediction capabilities of some sort. There is a great range of performance prediction methodologies. These can perhaps be most simply split into two categories:

- **history+heuristics**
  These methods are based on obtaining a performance history of the application on certain resources, and then using some heuristic to improve performance. This approach has been termed a "measure-modify" approach to performance tuning [4]. A more refined version of it is found in the "closed loop performance steering" design proposed by Reed and co-authors [17]. Measured performance "sensor" data are input to "decision procedures" to select resource management policies and enable "actuators" to improve performance. Off-line analysis of performance history data can identify improved decision procedures. This approach was taken further by the Autopilot system which used a fuzzy logic rule base for decision making [18].

- **parameters+model**
  Here an analytical performance model is derived which can be parameterised by application and system characteristics. The model is then used to generate performance predictions for specific application and system parameter values. Where system resource-related values are not constant, either monitored values or a range of values can be used [2][20]. The approach can be application-specific or based on simple models [21].

Note that in the history+heuristics methodology, there is no explicit performance prediction as such. The performance prediction appears to be implicit in the decision procedures and heuristics. The two approaches can perhaps be characterised (somewhat simplistically) as:

newSys&ApplParameters = heuristicPredictor (historyPerfData)
predictedPerfData = modelPredictor (sys&ApplParameters)

More generally, in an application which is composed, perhaps dynamically, of components (or, in the Open Grid Services Architecture (OGSA) [8], of services), a performance model *for the whole application* may not be available

or may be difficult to derive. On the other hand, the problem of forming and usefully applying heuristics on a complex component-based distributed application may be non-trivial. It is possible that data-mining of history data may provide a useful approach to this problem.

8.3 Hardware resource-usage data

In order to monitor the performance behaviour of the application components, it is necessary to have access to hardware resource-usage data. For example, if a component is not fully utilising all its resources, the steering system could choose to re-assign those unused resources to some other component, or even notify the external resource scheduler to free them. This resource-usage information could form part of the performance history data added to the performance repository.

There are general-purpose resource-usage monitoring and predicting services (e.g. [24]) which may be running on a computational Grid. There is an issue as to whether the performance steering system could or should interact with such general-purpose services. It may be sufficient to instrument the CPS, running on the same platform as the component, with calls to low-level monitoring facilities such as those available in the PAPI library [3].

On multi-user platforms, there is an issue of whether the performance steerer should take account of usage perturbations due to other applications running on the same platforms, or simply assume that the steered application has sole use of the resources. Indeed, there is a general and difficult problem facing the performance steerer: that of balancing the costs and benefits of re-distributing resources to components.

9. Generality of the performance steering system

9.1 Related work

A major aim of work on Grid and component framework middleware is to simplify the task of running applications on computational Grids, whilst at the same time maintaining performance QoS for individual applications. The ICENI project emphasises a component framework for generality and simplicity of use [9]. Application performance is achieved by an *application mapper* which selects the "best" component implementations for the resources available, based component meta-data. The GrADS project seeks to provide simplicity by building a framework for both preparing and executing Grid programmes [12]. Each application has an *application manager* which monitors the performance of that application for QoS achievement. Failure to achieve QoS contract causes a rescheduling or redistribution of resources. Both ICENI and GrADS have resource schedulers which partition resources between applications – described

in this paper as "external resource schedulers". GrADS monitors resources using NWS and uses Autopilot for performance prediction [24][18]. Such facilities will be included in ICENI in future [9].

The performance steering system being described in this paper makes no attempt at generality, except in its design. There is no scheduler which will distribute resources between applications - it is only concerned with single applications. In this it is similar to AppLeS [2]. It is intended to operate within such frameworks as GrADS and ICENI, providing perhaps finer-grain adaptive control over performance of the individual components, and potentially finer-grain control over application performance via the progress point model.

9.2 Grid Services

In the OGSA, components are regarded as providing services to other components within a framework which provides "dynamic discovery and composition of services" [8]. Presumably an OGSA service which is capable of being steered for performance will advertise itself as such, and will have performance steering facilities as part of its interface. This would allow some performance steering system to discover, connect with and steer that service. The design of the performance steering system being considered here should attempt to take account of this and related possibilities. Equally, it is possible for a performance-steered component or an entire application to act as an OGSA service, with performance steering not appearing in the service interface.

The Grid Monitoring Architecture (GMA), being developed by the Global Grid Forum Performance Working Group, similarly has registration and connection facilities for producers of performance data (such as hardware or software sensors) and consumers of performance-related information (such as a resource scheduler) via a directory service [22]. The performance steering system should also take account of the GMA. That is, the design should allow for the possibility of discovering and connecting dynamically to some performance data producer residing on one of the computing resources allocated to the steered application.

9.3 Interaction with Grid middleware

It cannot be assumed that future middleware resource managers will be able to interact with an application-specific performance steering system in sensible ways. An external resource manager which attempts to maintain some QoS for example, and which assumes it has sole rights to redistribute components over resources, is almost bound to conflict with a performance steering system which makes the same assumption (see section 4). Similar problems may be expected where an application is being steered simultaneously by a *application* steering system and by a *performance* steering system, as is the aim of the

RealityGrid project. There may be a problem for performance steering systems which arises implicitly from the very generality and flexibility necessarily built into Grid or component framework middleware. It would be expected that a performance steering system would be forced to emphasise efficiency at the cost of flexibility. For example, if information is available at link time or load time, then a performance steering system should incorporate it statically, rather than risking a loss of performance by obtaining the information dynamically at run time. Such inversion of aims may cause conflicts in the interaction of a performance steering system with generic middleware.

10. Current status and future work

There are a number of difficult questions which can best be resolved by actually carrying out implementation experiments. Ultimately a performance steering system must be about efficiency, which means reducing the costs of remote access, of processing data and of exerting control. Separation of concerns between APS and CPS is a valid design feature only if, in implementation, communication between APS and CPS is fast and infrequent. Similarly, a model of operation with fine-grain performance control is valid only if the fine-grain control-loop has low cost. Efficient ways of relaxing the synchronisation between CPS and APS at progress points required by the model of operation must be investigated. Although the performance steering system requires performance prediction and performance information repositories, the usefulness of the various approaches remains uncertain until they can be tested in an implementation.

An initial implementation of the design, based on a restricted set of requirements, is being carried out to investigate the design, and to clarify the interfaces between the entities in the performance steering system. A fixed set of resources are allocated to a component (a Fortran MPI implementation of a 3-D lattice-Boltzmann method[2] [14][15]) via a Globus RSL script, and the component loaders and APS are launched via *globusrun*. A component loader is a Unix process which launches the component and associated CPS. The CPS is a library linked to the component. There can be more than one component loader to allow the component, under control of the APS, to be migrated across machines. At present communication between APS and CPS, and between component loaders is via Unix sockets. Migration of the component utilises Globus Gridftp and FXDR [11][7]. The other entities in the design are to be added to this implementation of the basic design, in addition to investigating alternative implementation mechanisms.

[2] A "bottom up" three-dimensional lattice-Boltzmann method for the non-equilibrium and hydrodynamic simulation of binary immiscible and binary and ternary amphiphilic fluids

Acknowledgments

The work described in this paper has been carried out as part of the EPSRC-funded RealityGrid project, grant number GR/R67699.

References

[1] Atkinson, M.P., V. Dialani, L. Guy, I. Narang, N.W. Paton, D. Pearson, T. Storey and P. Watson (2002) *Grid Database Access and Integration: Requirements and Functionalities.* July 4, 2002 (draft).
http://www.cs.man.ac.uk/grid_db/documents.html

[2] Berman, F., R. Wolski, S. Figueira, J. Schopf, G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. *Proc. Supercomputing Conference*, 1996.

[3] Browne, S, J. Dongarra, N. Garner, G. Ho and P. Mucci. A portable programming interface for performance evaluation on modern processors. *Int J. of High Performance Computing Applications*, 14(3):189-204, 2000.

[4] Crovella, M.E. and T.J. LeBlanc. Parallel performance prediction using lost cycles analysis. *Proc. Supercomputing Conference*, 1994.

[5] Czajkowski, K., I. Foster and C. Kesselman. Resource co-allocation in computational Grids. *Proc. 8th Int. Symp. HPDC*, 219-228, 1999.

[6] Enterprise Java Beans. http://java.sun.com/products/ejb

[7] FXDR http://meteora.ucsd.edu/~pierce/fxdr_home_page.html

[8] Foster, I., C. Kesselman, J.M. Nick and S. Tuecke. *The Physiology of the Grid – An Open Grid Services Architecture for distributed systems integration*, 2002. http://www.globus.org

[9] Furmento, N., A. Mayer, S. McGough, S. Newhouse, T. Field and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Proc. Supercomputing Conference*, 2001.
http://www-icpc.doc.ic.ac.uk/components.

[10] Furmento, N., W. Lee, A. Mayer, S. Newhouse and J. Darlington. ICENI: An Open Grid Service Architecture Implemented with Jini. *Proc. Supercomputing Conference*, 2002.

[11] Globus Project. http://www.globus.org

[12] Kennedy, M., K., Mazina, J. Mellor-Crummey, K. Cooper, L. Torczon, F. Berman, A. Chen, H. Dail, O. Sievert, D. Angulo, I. Foster, D. Gannon, L. Johnsson, C. Kesselman, R. Aydt, D. Reed, J. Dongarra, S. Vadhiyar, R. Wolski. Toward a framework for preparing and executing adaptive grid programs. *Proc. Int. Parallel and Distributed Processing Symposium Workshop*, IEEE Computer Society Press, April, 2002.

[13] Nahrstedt, K. To overprovision or to share via QoS-aware resource management? *Proc. 8th Int. Symp. HPDC*, 205-212, 1999.

[14] Nekovee, M, and P. V. Coveney. Lattice-Boltzmann simulations of self-assembly of a binary water-surfactant system into ordered bicontinuous cubic and lamellar phases. *J. Am. Chem. Soc. 123*, 12380, 2001.

[15] Nekovee, M., J. Chin, N. Gonzlez-Segredo, and P. V. Coveney. A parallel lattice-Boltzmann method for large scale simulations of complex fluids. In E. Ramos et al. (Eds.), *Computational Fluid Dynamics, Proceedings of the Fourth UNAM Supercomputing Conference*, Singapore, World Scientific, 2001.

[16] RealityGrid Project. http://www.realitygrid.org

[17] Reed, D.A., C.L. Elford, T.M. Madhyastha, E. Smirni and S.E. Lamm. The next frontier: interactive and closed loop performance steering. *Proc 25th Annual Int Conf on Parallel Processing*, 20-31, 1996.

[18] Ribler, R.L, J.S. Vetter, H. Simitci and D.A. Reed. Autopilot: Adaptive control of distributed applications. *Proc. 7th Int. Symp. HPDC*, 172-179, 1998.

[19] Schopf, J.M. *A general architecture for scheduling on the Grid*, 2002. http://www-unix.mcs.anl.gov/~schopf/Pubs/jmspubs.html

[20] Schopf, J.M. and F. Berman. Performance prediction in production environments. *Proc. IPPS/SPDP*, 1998.

[21] Snavely, A, N. Wolter and L. Carrington. Modelling application performance by convolving machine signatures with application profiles. *IEEE 4th Annual Workshop on Workload Characterisation*, Austin, 2001. http://perc.nersc.gov/papers.html

[22] Tierney, B., R. Aydt, D. Gunter, W. Smith, M. Swany, V. Taylor and R. Wolski. *A Grid Monitoring Architecture.* Global Grid Forum Performance Working Group, 2002. http://www-didc.lbl.gov/GGF-PERF/GMA-WG

[23] Vetter, J. and K. Schwan. Techniques for high-performance computational steering. *IEEE Concurrency*, 7(4):63-74, 1999.

[24] Wolski, R., N.T. Spring and J. Hayes. The Network Weather Service: A distributed performance forecasting service for metacomputing. *J. Future Generation Computer Systems*, 15(5):757-768, 1999.

ADVANCES IN THE TAU PERFORMANCE SYSTEM

Allen D. Malony, Sameer Shende, Robert Bell, Kai Li, Li Li, Nick Trebon
Computer and Information Science Department
University of Oregon, Eugene, OR, USA
{malony,sameer,bertie,likai,lili,ntrebon}@cs.uoregon.edu

Abstract To address the increasing complexity in parallel and distributed systems and software, advances in performance technology towards more robust tools and broader, more portable implementations are needed. In doing so, new challenges for performance instrumentation, measurement, analysis, and visualization arise to address evolving requirements for how performance phenomena is observed and how performance data is used. This paper presents recent advances in the TAU performance system in four areas where improvements in performance technology are important: instrumentation control, performance mapping, performance interaction and steering, and performance databases. In the area of instrumentation control, we are concerned with the removal of instrumentation in cases of high measurement overhead. Our approach applies rule-based analysis of performance data in an iterative instrumentation process. Work on performance mapping focuses on measuring performance with respect to dynamic calling paths when the static callgraph cannot be determined prior to execution. We describe an online performance data access, analysis, and visualization system that will form the basis of a large-scale performance interaction and steering system. Finally, we describe our approach to the management of performance data in a database framework that supports multi-experiment analysis.

Keywords: performance tools, parallel computing, distributed computing

1. Introduction

There has long been a tension in the field of performance tools research between the need to invent new techniques to deal with performance complexity of next-generation parallel and distributive systems, and the need to develop tools that are robust, both in their function and in their scope of application. Perhaps this is just the nature of the field. Yet there are important issues concerning the advancement of performance tools "research" and the successful demonstration and use of performance "technology." A cynical perspective might argue against trying something new without first getting the existing technol-

ogy to just work, and work reliably in real applications and environments. The all too commonly heard mantras "performance tools don't work" and "performance tools are too hard to use" might lead one to believe in this perspective, but research history does not necessarily justify such a strong cynical stance. There has been significant innovation in performance observation and analysis techniques in the last twenty year to address the new performance challenges parallel computing environments present [10]. Current attention is certainly being paid to easing the burden of tool use through automated analysis [1]. There have also been important technology developments that add considerable value to the performance tool repertoire, such as APIs for dynamic instrumentation [4] and hardware performance counters [3]. Why, then, is there an apparent disconnect between research results and the "reality" of tool usage in parallel application environments?

From our perspective as performance tool researchers, we take, perhaps, a controversial stance among our peers and argue that tool engineering is an important factor in this regard. The controversial part primarily concerns the notion of "research" and the rewards (or lack thereof) in a research career for tool development. Our counter position is that innovation in performance tools research is best advanced by "standing on the shoulders" of solid technology foundations. When that foundation does not exist, it must be developed. When a technology does exist, it should be integrated, if possible, and not reinvented. Indeed, many tools do not work reliably and, as a consequence, are hard to use. Many tools are not portable across parallel systems or reusable with different programming paradigms, and, as a consequence, have limited application. These results cannot be considered as positive results for the performance tool research community, that is, if reliability, portability, and robustness, in general, is considered worthy of research. We believe that they are, particularly in parallel computing. Furthermore, we contend that the future advances in performance tools research with the most direct potential effect in real application will be those that can best leverage and amplify existing robust performance technology.

In this paper, we consider four research problems being investigated in the TAU parallel performance system [9, 17] and describe the performance tools being developed to address them. These tools build on and leverage the capabilities in TAU (as well as the other technologies integrated in TAU) to provide robust, value-added solutions. While none of these solutions are necessarily "new," in the sense of a new research finding, the technology being developed is novel and will directly provide new capabilities to TAU users. After a brief description of the TAU performance system, we look at the problem of instrumentation control to reduce measurement overhead. Our work here builds on TAU's rich instrumentation framework. The second problem of callpath profiling requires a solution that maps performance measurements to dynamically

occurring callpaths. Here, TAU's performance mapping API is utilized. Providing online performance analysis and visualization for large-scale parallel applications is the third problem we consider. Finally, we describe our early work to develop a performance database framework that can support multi-experiment performance analysis.

2. TAU Performance System

For the past twelve years, the TAU project has conducted research on performance tools for parallel and distributed systems. The goal of this work has mainly been the development of robust technology to meet evolving performance evaluation challenges of state-of-the-art parallel systems and applications. In particular, we have focused on problems of performance tool portability, extendability, and interoperation.

The TAU performance system [9, 17] is our integrated toolkit for performance instrumentation, measurement, analysis, and visualization of large-scale parallel applications. It targets a general computation model consisting of shared-memory computing *nodes* where *contexts* reside, each providing a virtual address space shared by multiple *threads* of execution. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Because TAU enables performance information to be captured at the *node/context/thread* levels, this information can be mapped to the particular parallel software and system execution platform under consideration.

As shown in Figure 1, the TAU system supports a flexible instrumentation model that applies at different stages of program compilation and execution. The instrumentation targets multiple code points, provides for mapping of low-level execution events to higher-level performance abstractions, and works with multi-threaded, message passing, and mixed-mode parallel computation models. Different instrumentation techniques are supported, including dynamic instrumentation using the DyninstAPI [4]. All instrumentation code makes calls to the TAU measurement API to provide a common measurement model. The TAU measurement library implements performance profiling and tracing support for performance events occurring at function, method, basic block, and statement levels. Performance experiments can be composed from different measurement modules (e.g., hardware performance monitors, such as PAPI [3]) and measurements can be collected with respect to user-defined performance groups. C, C++, Fortran 77/90, OpenMP, and Java languages are supported. The TAU data analysis and presentation utilities offer text-based and graphical tools to visualize the performance data as well as bridges to third-party software, such as Vampir [11] and Paraver [12] for sophisticated trace analysis and visualization.

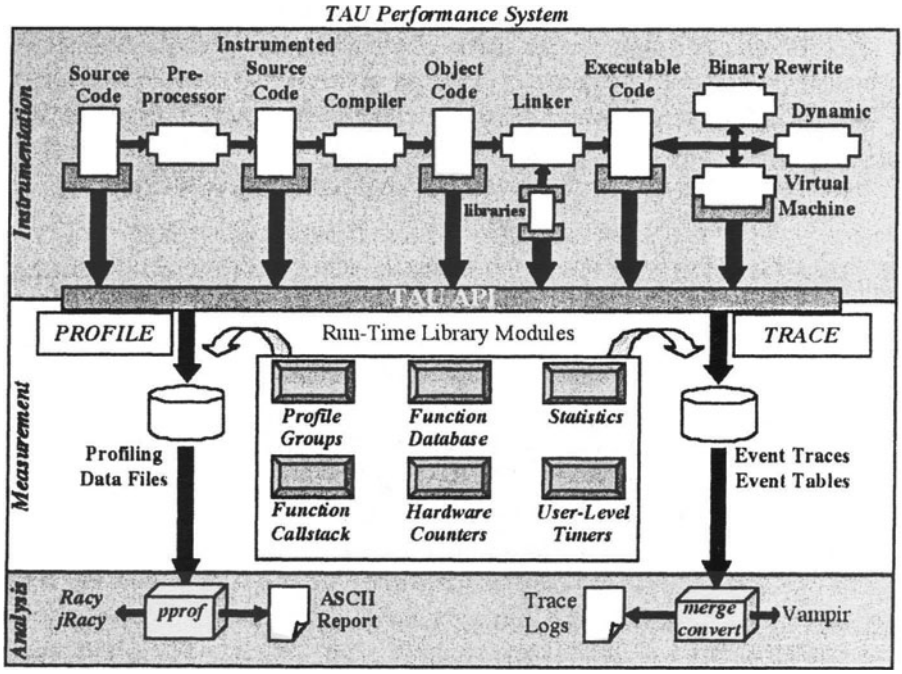


Figure 1. Architecture of the TAU performance system.

TAU has been ported to nearly all high-performance computing platforms and is being used extensively in the performance analysis of DOE applications. TAU is also being applied as the primary performance technology across a diverse set of code development projects, including Uintah [15], CCA [2], VTF [18], and SAMRAI [7]. Although the current set of features in the TAU performance system is quite substantial, it is important to note that users are always requesting new capabilities. The interesting research problems that arise concern how to develop new techniques to address these requests while maintaining tight integration with the rest of the TAU system. The four problems below are all representative of such endeavors.

3. Measurement Overhead and Instrumentation Control

The selection of what "events" to observe when measuring the performance of a parallel application is an important consideration, as it is the basis for how performance data will be interpreted. The performance events of interest depend mainly on what aspect of the execution the user wants to see, so as to construct a meaningful performance view from the measurements made. Typical events include control flow events that identify points in the program that are executed, or operational events that occur when some operation or action has

been performed. Events may be atomic or paired to mark begin and end points, for example, to mark the entry and exit of a routine. Choice of performance events also depends on the scope and resolution of the performance measurement desired. However, the greater the degree of performance instrumentation in a program, the higher the likelihood that the performance measurements will alter the way the program behaves, an outcome termed *performance perturbation.*

Most performance tools, including TAU, address the problem of performance perturbation indirectly using techniques to reduce the performance intrusion (i.e., overhead) associated with performance measurement. This overhead is a result of two factors: 1) the execution time to make the measurement relative to the "size" of the event, and 2) the frequency of event occurrence. The first factor concerns the influence of the measurement overhead on the observed performance of a *particular* event. If the overhead is large relative to the size of the event, the performance measurement is less likely to be accurate unless its overhead is compensated in some way. The overhead is typically measured in execution time, but can also include the impact on other metrics, such as hardware counts. The second factor relates to overheads as seen from the perspective of the entire program. That is, the higher the frequency of events, the larger percentage of the execution will be taken up by performance measurement.

Techniques to control performance intrusion are directed towards making performance measurement more efficient or controlling the performance instrumentation. The former is a product of engineering of the measurement system. That is, the lighter-weight the measurment system, the lower the overhead. Here, we are concerned with controlling performance instrumentation to remove or disable performance measurement for "small" events or events that occur with high frequency. Clearly this will eliminate the overhead otherwise generated, but how are these events determined before a measurment is made? It may be possible for sophisticated source code analysis to identify small code segments, but this is not a complete solution since the execution time could depend on runtime parameters. Plus, we would like a solution to work across languages and few static analysis tools are available.

Instead, a direct measurement approach will likely be needed. The idea is that a series of instrumentation experiments would be conducted to observe the measurement overhead, weeding out those events resulting in unacceptable levels of intrusion. Whereas this performance data analysis and instrumentation control can be done manually, it is tedious and error-prone, especially when the number of performance events is large. Thus, the problem we addressed was how to develop a tool to help automate the process in TAU.

The TAU performance system instruments an application code using an optional *instrumentation control file* that identifies events for inclusion and exclu-

sion. The TAU instrumentor's default behavior is to instrument every routine and method. Obviously, this instrumentation may result in high measurement overhead, and the user can manually modify the file to eliminate small events, or those that are not interesting to observe. As noted above, this is a cumbersome process. Instead, the *TAUreduce* tool allows the user to write instrumentation rules that will be applied to the parallel measurement data to identify which events to exclude. The output of the tool is a new instrumentation control file with those events de-selected for instrumentation, thereby reducing measurement overhead in the next program run.

Table 1 shows examples of the *TAUreduce* rule language. A *simple rule* is an arithmetic condition written as:

[EventName: | GroupName:] Field Operator Number

where *Field* is a TAU profile metric (e.g., numcalls, percent, usec, usec/call), *Operator* is one of <, >, or =, and *Number* is any number. A rule applies to all events unless specified explicitly, either by the *EventName* (e.g., routineA) or by the event *GroupName* (e.g., TAU_USER). In the latter case, all events that belong to the group are selected by the rule. A *compound rule* is a logical conjunction of simple rules. Multiple rules, appearing on separate lines, are applied disjunctively.

Description	Rule
Exclude all events that are members of the TAU_USER group and use less than 100 microseconds	$TAU\_USER : usec < 100$
Exclude all events that have less than 100 microseconds and are called only once	$usec < 100 \ \& \ numcalls = 1$
Exclude all events that have less than 100 microseconds per call or have a total inclusive percentage less than 5	$usecs/call < 100$ $percent < 5$

Table 1. Examples of TAUreduce rule language.

As a simple example of applying the instrumentation reduction analysis, consider two algorithms to find the kth largest element in a list of N unsorted elements. The first algorithm (*kth_largest_qs*) uses *quicksort* first to sort the list and then selects the kth element. The second algorithm (*select_kth_largest*) scans the list keeping a sorted set of the k largest elements seen thus far. At the end of the scan, it selects the least of the set. We ran the program on a list of 1,000,000 elements with k=2324, first with minimal instrumentation to determine the execution time of the two algorithms: *kth_largest_qs* (.188511 secs), *select_kth_largest* (.149594 secs). Total execution time was .36 secs on a 1.2 Ghz Pentium III machine.

Then the code was instrumented fully and run again. The profile results are shown in the top half of Figure 2. Clearly, there is significant performance overhead and the execution times are not accurate, even though TAU's per event measurement overhead is very low. We defined the rule

$$usec > 1000 \ \& \ numcalls > 400000 \ \& \ usecs/call < 30 \ \& \ percent > 25$$

and applied *TAUreduce*, eliminating the events marked with "(*)". Running the code again produced the results in the lower half of Figure 2. As seen, the execution times are closer to what we expect.

```
NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec        msec                           usec/call
---------------------------------------------------------------------------------------
100.0          13        4,982           1           4    4982030 int main
 93.5       3,223        4,659 4.20241E+06 1.40268E+07          1 void quicksort (*)
 62.9     0.00481        3,134           5           5     626839 int kth_largest_qs
 36.4         137        1,813          28      450057      64769 int select_kth_largest
 33.6         150        1,675      449978      449978          4 void sort_5elements (*)
 28.8       1,435        1,435 1.02744E+07           0          0 void interchange (*)
  0.4          20           20           1           0      20668 void setup
  0.0      0.0118       0.0118          49           0          0 int ceil

NODE 0;CONTEXT 0;THREAD 0:
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                           usec/call
---------------------------------------------------------------------------------------
100.0          14          383           1           4     383333 int main
 50.9         195          195           5           0      39017 int kth_largest_qs
 40.0         153          153          28          79       5478 int select_kth_largest
  5.4          20           20           1           0      20611 void setup
  0.0        0.02         0.02          49           0          0 int ceil
```

Figure 2. Example application of *TAUreduce* tool.

While the above example is rather simple, the *TAUreduce* tool can be applied to large parallel applications. It is currently being used in Caltech's ASAP ASCI project to control instrumentation in the Virtual (Shock) Test Facility (VTF) [18]. *TAUreduce* is part of the TAU performance system distribution and, thus, is supported on all platforms where TAU is available. It is currently being upgraded to include analysis support for multiple performance counters.

One important comment about this work is that it deals with a fundamentally practical problem in parallel performance observation, that is, the tradeoff of measurement detail and accuracy. By eliminating events from instrumentation, we lose the ability to see those events at all. If the execution of small routines accounts for a large portion of the execution time, that may be hard to discern without measurement. On the other hand, accurate measurement is confounded by high relative overheads. We could attempt to track these overheads at run-time and subtract accummulated overhead when execution time measurements

are made. This is something we are pursuing in TAU to increase timing accuracy, but it requires determining a minimum overhead value from measurement experiments on each target platform. Another avenue is to change performance instrumentation on-the-fly as the result of identifying high overhead events. We are also considering this approach in TAU.

4. Performance Mapping and Dynamic Callpath Profiling

To observe meaningful performance events requires placement of instrumentation in the program code. However, not all information needed to interpret an event of interest is available prior to execution. A good example of this occurs in callgraph profiling. Here the objective is to determine the distribution of execution time along the dynamic routine calling paths of an application. A callpath of length k is a sequence of the last $k-1$ routines called. To measure execution time spent on a callpath requires identifying the begin and end points during which a callpath is "active." These points are just the entry and exit of a called routine. If $k = 1$, callpath profiling is the measurement of amount of time spent in a routine for each of its calling parents. The basic problem with callpath profiling is that the identities of all k-length calling paths ending at a routine may not, and generally are not, known until the application finishes its execution. How, then, do we identify the dynamic callpath events in order to make profile measurements?

One approach is not to try to not identify the callpaths at runtime, and instead instrument just basic routine entry and exit events and record the events in a trace. Trace analysis can then easily calculate callpath profiles. The problem, of course, with this approach is that the trace generated may be excessively large, particularly for large numbers of processors. Unfortunately, the instrumentation and measurement problem is significantly harder if callpath profiles are calculated online.

If the whole source is available, it is possible to determine the entire static callgraph and enumerate all possible callpaths, encoding this information in the program instrumentation. These callpaths are static, in the sense that they could occur; dynamic callpaths are the subset of static callpaths that actually do occur during execution. Once a callpath is encoded and stored in the program, the dynamic callpath can then be determined directly by indexing a table of possible next paths using the current routine id. Once the callpath is known, the performance information can be easily recorded in pre-reserved static memory. This technique was used in the CATCH tool [5].

Unfortunately, this is not a robust solution for several reasons. First, source-based callpath analysis is non-trivial and may be only available for particular source languages, if at all. Second, the application source code must be available if a source-based technique is used. Third, static callpath analysis is possible

at the binary code level, but the routine calls must be explicit and not indirect. This complicates C++ callpath profiling, for instance.

To deliver a robust, general solution, we decided to pursue an approach where the callpath is calculated and queried at runtime. The TAU measurement system already maintains a callstack that is updated with each entry/exit performance event (e.g., routine entry and exit). Thus, to determine the k-length callpath when a routine is entered, all that is necessary is to traverse up the callstack to determine the last $k - 1$ events that define the callpath. If this is a newly encountered callpath, a new measurement profile must be created at that time because it was not pre-allocated. The main problem is how to do all of this efficiently.

Mapping callpath identity to its profile measurement is an example of what we call *performance mapping*. TAU implements a performance mapping API based on the *Semantic Event Association and Attribute* (SEAA) model [14]. Here an association is built between the identity of a performance event (e.g., a callpath) and a performance measurement object (e.g., a profile) for that event. Thus, when the event is encountered, the measurement object linked to that event can be found, via a lookup in a mapping table, and the measurement made.

In the case of callpath performance mapping, new callpaths occur dynamically, requiring new profile objects to be created at runtime. This can be done efficiently using the TAU mapping API. The callpath name is then hashed to serve as the index for future reference. Because routine identifiers can be long strings, TAU optimizes this process by computing the hash based on addresses of the profile objects of its $k - 1$ parents. While the extra overhead to perform this operation is fixed, the accumulated overhead will depend on the number of unique k-length callpaths encountered in the computation, as each of these will need a separate profile object created.

We have implemented 2-level callpath profiling in TAU as part of the current TAU performance system distribution. The important result is that this capability is available in all cases where TAU profiling is available. It is not restricted by programming language, nor source code access required, as dynamic instrumentation (via DyninstAPI [4]) can be used when source is not available. Also, all types of performance measurements are allowed, including measuring hardware counts for each callpath. Finally, in the future, we can benefit from the overhead reduction mechanisms to eliminate particular callpaths from measurement consideration.

Unfortunately, unlike a static approach, the measurement overhead of this dynamic approach increases as k increases because we must walk the callstack to determine the callpath. We have discussed several methods to do this more efficiently, but none lead to a fixed overhead for any k, and adopting a general k solution for the 2-level case would result in greater cost. Most user requests

were for 2-level callpaths to determine routine performance distribution across calling parents, and this is what has been implemented in TAU. It should be noted that there are no inherent limitations to implementing solutions with $k > 2$. Also, if it is possible to determine callpaths statically, TAU could certainly use that information to implement a fixed-cost solution.

5. Large-Scale Performance Monitoring and Steering

Parallel performance tools offer the program developer insights into the execution behavior of an application. However, most tools do not work well with large-scale parallel applications where the performance data generated comes from thousands of processes. Not only can the data be difficult to manage and the analysis complex, but existing performance display tools are mostly restricted to two dimensions and lack the customization and interaction to support full data investigation. In addition, it is increasingly important that performance tools be able to function online, making it possible to control and adapt long-running applications based on performance feedback. Again, large-scale parallelism complicates the online access and management of performance data. It may be desirable to use existing computational steering systems for this purpose, but this will require performance analysis and visualization to be integrated with these tools.

As a result of our work with the University of Utah [16], we found ourselves in a position to design and prototype a system architecture for coupling advanced three-dimensional visualization with online performance data access, analysis, and visualization in a large-scale parallel environment. The architecture, shown in Figure 3, consists of four components. The "performance data integrator" component is responsible for interfacing with a performance monitoring system to merge parallel performance samples into a synchronous data stream for analysis. The "performance data reader" component reads the external performance data into internal data structures of the analysis and visualization system. The "performance analyzer" component provides the analysis developer a programmable framework for constructing analysis modules that can be linked together for different functionality. The "performance visualizer" component can also be programmed to create different displays modules.

Our prototype is based on the TAU performance system, the Uintah computational framework [15], and the SCIRun [13] computational steering and visualization system. Parallel profile data from a Uintah simulation are sampled and written to profile files during execution. The performance data integrator reads the performance profile files, generated for each profile sample for each thread, and merges the files into a single, synchronized profile sample dataset. Each profile sample file is assigned a sequence number and the whole dataset is sequenced and timestamped. A socket- based protocol is maintained with the

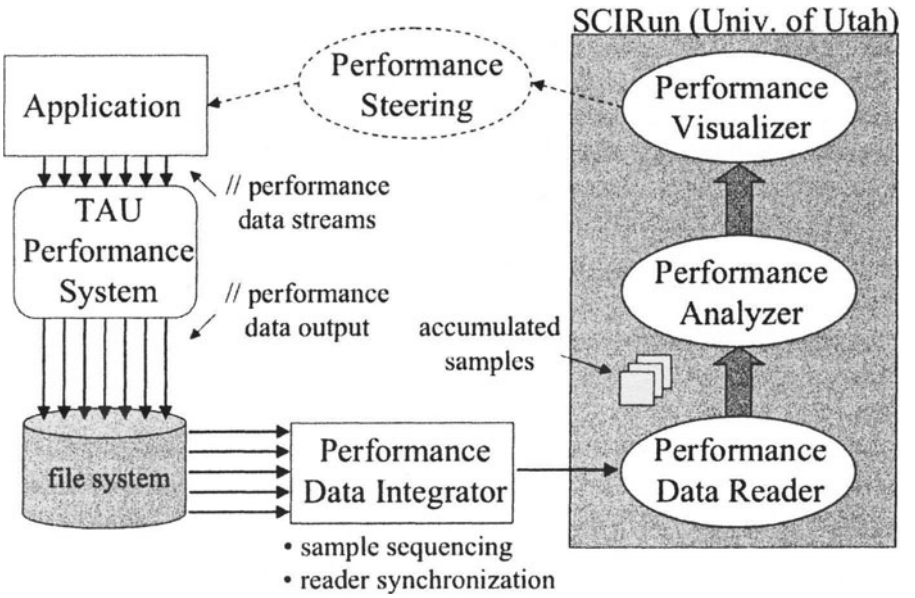


Figure 3. Online performance analysis and visualization architecture.

performance data reader to inform it of the availability of new profile samples and to coordinate dataset transfer.

The performance profile reader, implemented as a SCIRun module, inputs the merged profile sample dataset sent by the data integrator and stores the dataset in an internal C++ object structure. A profile sample dataset is organized in a tree-like manner according to TAU profile hierarchy:

$$node \rightarrow context \rightarrow thread \rightarrow profile\ data$$

Each object in the profile tree has a set of attribute access methods and a set of offspring access methods.

Using the access methods on the profile tree object, all performance profile data, including cross-sample data, is available for analysis. Utah's SCIRun [13] provides a programmable system for building and linking the analysis and visualization components. A library of performance analysis modules can be developed, some simple and others more sophisticated. We have implemented two generic profile analysis modules: *Gen2DField* and *Gen3DField*. The modules provide user control that allows them to be customized with respect to events, data values, number of samples, and filter options. Ultimately, the output of the analysis modules must be in a form that can be visualized. The *Gen2DField* and *Gen3DField* modules are so named because they produce 2D and 3D *Field* data, respectively. SCIRun has different geometric meshes

available for Fields. We use an *ImageMesh* for 2D fields and a *PointCloudMesh* for 3D fields.

The role of the performance visualizer component is to read the Field objects generated from performance analysis and show graphical representations of performance results. We have built three visualization modules to demonstrate the display of 2D and 3D data fields. The *Terrain* visualizer shows ImageMesh data as a surface. The user can select the resolution of the X and Y dimensions in the Terrain control panel. A *TerrainDenotator* module was developed to mark interesting points in the visualization. A different display of 2D field data is produced by the *KiviatTube* visualizer. Here a "tube" surface is created where the distance of points from the tube center axis is determined by metric values and the tube length correlates with the sample. The visualization of PointCloudMesh data is accomplished by the *PointCloud* visualizer module.

The SCIRun program graph in Figure 4 shows how the data reader, analyzer, and visualizer modules are connected to process parallel profile samples from a Uintah application. The visualization is for a 500 processor run and shows the entire parallel profile measurement. The performance events are along the left-right axis, the processors along the in-out axis, and the performance metric (in this case, the exclusive execution time) along the up-down axis. Denotators are used to identify the performance events in the legend with the largest metric values. This full performance view enables the user to quickly identify major performance contributors.

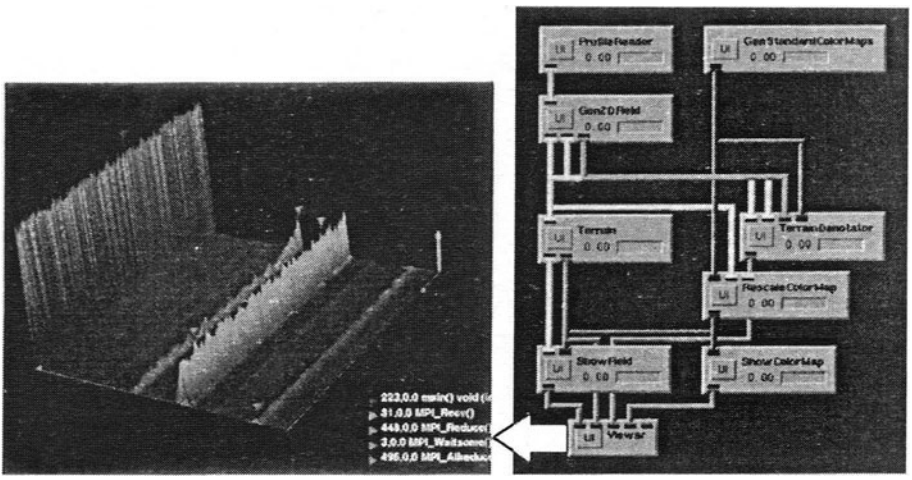


Figure 4. Performance profile visualization of 500 Uintah processes.

Although this work is in the early stages, it demonstrates the significant tool advances possible through technology integration. As the Utah C-SAFE ASCI project moves towards Uintah computations with adaptive-mesh refinement

capabilities, we expect the relevance of online performance analysis to increase in importance. We are developing new performance visualization modules and extending the performance profile data to accommodate hardware counter statistics. Since SCIRun is being positioned as a computational steering system for Uintah, the implementation of the online performance tool in SCIRun well positions it for use as a customizable performance steering tool.

6. Peformance Database Framework

Empirical performance evaluation of parallel and distributed systems often generates significant amounts of performance data and analysis results from multiple experiments as performance is being investigated and problems diagnosed. Yet, despite the broad utility of cross-experiment performance analysis, most current performance tools support performance analysis for only a single application execution. We believe this is due primarily to a lack of tools for performance data management. Hence, there is a strong motivation to develop performance database technology that can provide a common foundation for performance data storage and access. Such technology could offer standard solutions for how to represent the performance data, how to store them in a manageable way, how to interface with the database in a portable manner, and how to provide performance information services to a broad set of analysis tools and users. A performance database system built on this technology could serve both as a core module in a performance measurement and analysis system, as well as a central repository of performance information contributed to and shared by several groups.

To address the performance data management problem, we designed the Performance DataBase Framework (PerfDBF) architecture shown in Figure 5. The PerfDBF architecture separates the framework into three components: performance data input, database storage, database query and analysis. The performance data is organized in a hierarchy of *applications*, *experiments*, and *trials*. Application performance studies are seen as constituting a set of experiments, each representing a set of associated performance measurements. A trial is a measurement instance of an experiment. We designed a Performance Data Meta Language (PerfDML) and PerfDML translators to make it possible to convert raw performance data into the PerfDB internal storage. The Performance DataBase (PerfDB) is structured with respect to the *application/experiment/trial* hierarchy. An object-relational DBMS is specified to provide a standard SQL interface for performance information query. A Performance DataBase Toolkit (PerfDBT) provides commonly used query and analysis utilities for interfacing performance analysis tools.

To evaluate the PerfDBF architecture, we developed a prototype for the TAU performance system for parallel performance profiles. The prototype PerfDBF

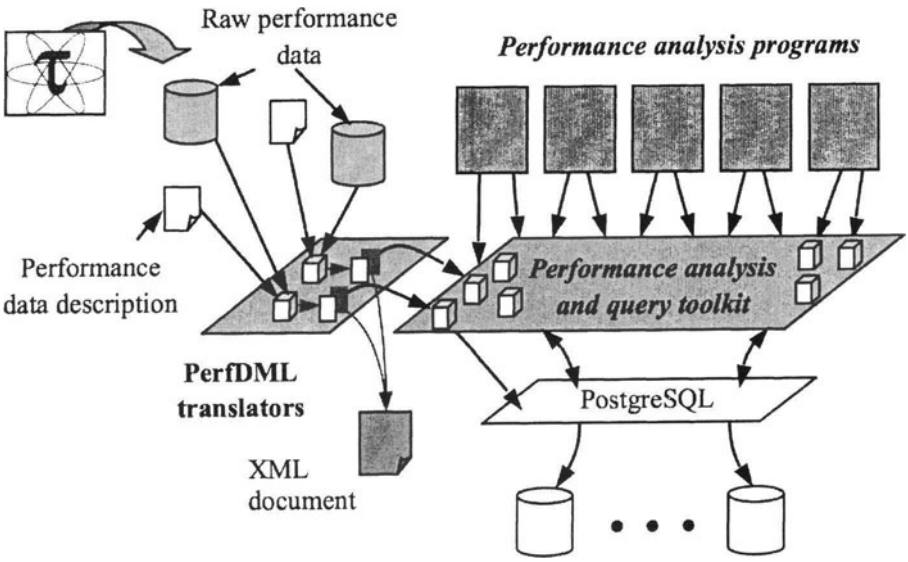


Figure 5. TAU performance database framework

converts the raw profiles to PerfDML form, which is realized using XML technology. Database input tools read the PerfDML documents and store the performance profile information in the database. Analysis tools then utilize the PerfDB interface to perform intra-trial, inter-trial, and cross-experiment query and analysis. To demonstrate the usefulness of the PerfDBF, we have developed a scalability analysis tool. Given a set of experiment trials, representing execution of a program across varying numbers of processors, the tool can compute scalability statistics for every routine for every performance metric. As an extension of this work, we are applying the PerfDBF prototype in a performance regression testing system to track performance changes during software development.

The main purpose of the PerfDBF work is to fill a gap in parallel performance technology that will make it possible for performance tools to interoperate. The PPerfDB [8] and Aksum [6] projects have demonstrated the benefit of providing such technology support and we have hopes to merge our efforts. We already see benefits within the TAU toolset. Our parallel performance profile, ParaProf, is able to read profiles that are stored in PerfDBF. In general, we believe the key will be to find common representations of performance data and database interfaces that can be adopted as the lingua franca among performance information producers and consumers. Its implementation will be an important enabling step forward in performance tool research.

7. Conclusions

The research work we presented in this paper reflects our view that advances in performance technology will be a product of both innovative ideas and strong engineering. More importantly, as the performance complexity of parallel and distributed systems increases, it will be important to develop performance tools on a robust technology foundation, leveraging existing capabilities to realize more sophisticated functionality. We believe the tools described above demonstrate this result. Each is or will be implemented in a form that can be distributed with the TAU performance system. While this may go beyond what is necessary to "prove" a research result, it is in the application of a performance tool on real performance problems where its merit will be truly determined.

References

[1] APART, IST Working Group on Automatic Performance Analysis: Real Tools. See http://www.fz-juelich.de.

[2] R. Armstrong, et al., Toward a Common Component Architecture for High-Performance Scientific Computing, High Performance Distributed Computing Conference, 1999. See http://www.cca-forum.org.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, A Portable Programming Interface for Performance Evaluation on Modern Processors, *International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[4] B. Buck and J. Hollingsworth, An API for Runtime Code Patching, *International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[5] Luiz DeRose and Felix Wolf, CATCH - A Call-Graph Based Automatic Tool for Capture of Hardware Performance Metrics for MPI and OpenMP Applications, Euro-Par 2002, pp. 167–176.

[6] T. Fahringer and C. Seragiotto, Experience with Aksum: A Semi-Automatic Multi-Experiment Performance Analysis Tool for Parallel and Distributed Applications, Workshop on Performance Analysis and Distributed Computing, 2002.

[7] R. Hornung and S. Kohn, Managing Application Complexity in the SAMRAI Object-Oriented Framework, *Concurrency and Computation: Practice and Experience*, special issue on Software Architectures for Scientific Applications, 2001.

[8] K. Karavanic, PPerfDB. See http://www.cs.pdx.edu/ karavan/research.htm.

[9] A. Malony and S. Shende, Performance Technology for Complex Parallel and Distributed Systems, in *Distributed and Parallel Systems From Instruction Parallelism to Cluster Computing*, G. Kotsis and P. Kacsuk (Eds.), Kluwer, pp. 37–46, 2000.

[10] A. Malony, Tools for Parallel Computing: A Performance Evaluation Perspective, in Handbook on Parallel and Distributed Processing, J. Blazewicz, K. Ecker, B. Plateau, an D. Trystram (Eds.), 2000, Springer-Verlag, pp. 342–363.

[11] W.Nagel, A.Arnold, M.Weber, H.Hoppe, and K.Solchenbach, Vampir: Visualization and Analysis of MPI Resources, *Supercomputing*, 12(1):69–80, 1996.

[12] Paraver, European Center for Parallelism of Barcelona, Technical University of Catalonia. See http://www.cepba.upc.es/paraver/index.html.

[13] S. Parker, D. Weinstein, and C. Johnson, The SCIRun Computational Steering Software System, in *Modern Software Tools in Scientific Computing*, E. Arge, A. Bruaset, and H. Langtangen (Eds.), Birkhauser Press, pp. 1–44, 1997.

[14] Sameer S. Shende, The Role of Instrumentation and Mapping in Performance Measurement, PhD Thesis, University of Oregon, 2001.

[15] J. St. Germain, J. McCorquodale, S. Parker, and C. Johnson, Uintah: A Massively Parallel Problem Solving Environment, *Proceedings of HPDC 2000*, pp. 33–41, 2000.

[16] J. St. Germain, A. Morris, S. G. Parker, A. D. Malony, and S. Shende, Integrating Performance Analysis in the Uintah Software Development Cycle, International Symposium on High Performance Computing, pp. 190–206, 2002.

[17] TAU (Tuning and Analysis Utilities). See http://www.acl.lanl.gov/tau.

[18] VTF, Virtual Test Shock Facility, Center for Simulation of Dynamic Response of Materials. See http://www.cacr.caltech.edu/ASAP.

UNIFORM RESOURCE VISUALIZATION: SOFTWARE AND SERVICES

Kukjin Lee, Diane T. Rover
Electrical and Computer Engineering
Iowa State University, IA, USA
{leekukji,drover}@iastate.edu

Abstract Computing environments continue to increase in scale, heterogeneity, and hierarchy, with resource usage varying dynamically during program execution. Computational and data grids and distributed collaboration environments are examples. To understand performance and gain insights into developing applications that efficiently use the system resources, performance visualization has proven useful. However, visualization tools often are specific to a particular resource or level in the system, possibly with fixed views, and thus limit a user's ability to observe and improve performance. Information integration is necessary for system-level performance monitoring. Uniform resource visualization (URV) is a component-based framework being developed to provide uniform interfaces between resource instrumentation, called resource monitoring components (RMC) and performance views, called visualization components (VC). URV supports services for connecting VCs to RMCs, and creating multi-level views, as well as visual schema definitions for sharing and reusing visualization design knowledge.

Keywords: performance visualization, visualization design knowledge, reusable visualization, resource monitoring, heterogeneous systems, Grid computing

1. Introduction

Visualization has been widely accepted as a means to deal with large-scale data sets including performance events describing dynamic and multivariate behavior of computer systems. Many performance visualization techniques have been developed and applied to complex parallel and distributed systems. Most of them, however, have focused on specific types of events at a certain level in the system and have been developed in isolation. In contrast, a distributed application in heterogeneous environments (such as grids) produces various types of events at multiple levels during execution and often needs to be dynamically configured for newly available resources.

Monitoring such applications is not trivial. It requires an open monitoring architecture for accessing distributed performance events, systematic methods for event correlation across semantic levels, as well as scalable visualization for dynamic creation of performance views [3]. Several monitoring frameworks [14][16][19][23] have been developed to provide open architectures and data correlation. However, visualizations still rely primarily on domain- and resource-specific tools that are neither scalable nor interoperable. Performance visualization technology should scale with system complexity, such that what is viewed and how it is viewed are not constants. The technology should be reusable and interoperable to capture performance problems associated with various resources across domains.

We have been exploring a new monitoring framework called Uniform Resource Visualization, URV, for constructing visualizations and for monitoring and analyzing complex parallel/distributed systems. It is designed to allow different resources, at different levels, to be viewed and interacted with in a consistent and coordinated manner. URV addresses the following needs in the visualization of distributed systems: (1) the need for applying standard visualization services (uniformity); (2) the need for composing system-level views (composition); and (3) the need for sharing visualization design knowledge (reusability).

Uniformity in URV is implemented by bridging one or more resources to a visualization with interoperable interfaces and by describing resources and visualizations with descriptors. These mechanisms provide users with uniform interfaces for accessing, viewing, and managing heterogeneous resources. Composition refers to support for higher level, e.g., system, visualization. Reusability in URV is provided via visual design knowledge representation and discovery. Users can retrieve existing visualizations, and by composing them, can construct a new visualization. Distinctive services in URV include: (1) connection service to bind resource(s) to a visualization; and (2) reuse service to share visual design knowledge and to construct a new visualization by adapting the knowledge.

2. Visualization Model

A visualization in URV, termed a URV view [25], is associated with one or more resources, where a resource may be a physical entity (e.g., router) or a logical entity (e.g., process). It consists of a resource part and a visualization part, as illustrated in Figure 1. The resource part of a URV view consists of a resource and a resource-monitoring component (RMC). An RMC provides performance monitoring and control services; it separates a visualization from the physical or logical resource being visualized. The visualization part of a URV view is termed a visualization component (VC). Each component spec-

ifies services comprised in its interface to other components, e.g., r1 r2 . . . rn and v1 v2 . . . vn in Figure 1. An interface, called a connector, defines component interaction via a set of services that are provided by one component and required by another component. In order to create components (e.g., RMC, VC), we can convert existing tools or software modules by wrapping them with URV application programming interfaces (APIs). For instance, network sensors in NWS [16] and monitoring tools in Netlogger [15] can be wrapped to create RMCs, and an nlv visualization [38] can be converted to a VC. To

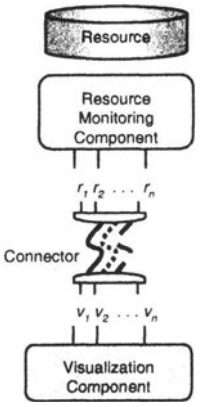


Figure 1. URV view and its elements.

represent a URV view uniformly, we describe RMCs and VCs with formal descriptors. Descriptors, in general, allow a designer to specify the interfaces to components rigorously and to specify component interconnection at these interfaces. An ideal description of a component encompasses the component's concept, content, and context [24]. Concept is a description of what the component does, including its interface and any operating specifications. Content describes how the concept is realized or implemented so that users of the component can modify or adapt it to a specific use. Finally, context describes the domain of applicability of the component. For instance, in a VC descriptor, the attributes of the data model and interface describe the component's concept, and the attributes of the graphical representation and the layout, its content. The content is essential to the creation of rules for visualization composition, which is based on shared attributes of graphical representations [9]. The attributes of component identification (e.g., name, location) denote the component's context. With this approach, components may be specified uniformly and be reusable and interoperable.

URV views can be implemented by mapping onto existing software architectures, e.g., the Grid Monitoring Architecture (GMA) [21] depicted in Figure 2(a). GMA is a new performance monitoring architecture in Grid computing that has been proposed by a Global Grid Forum (GGF) [40] Performance group. The basic model of GMA consists of three elements: a directory service, a producer, and a consumer. A directory service supports registry and discovery of performance events. A producer provides performance data, and a consumer uses the performance data for analysis. GMA is simple and scalable. As depicted in Figure 2(b), an RMC and a VC in URV correspond to a producer and a consumer in GMA, respectively, in the sense that the VC consumes event data that the RMC produces. A GMA-based monitoring tool could include selected URV components, such as a specific VC for visualization.

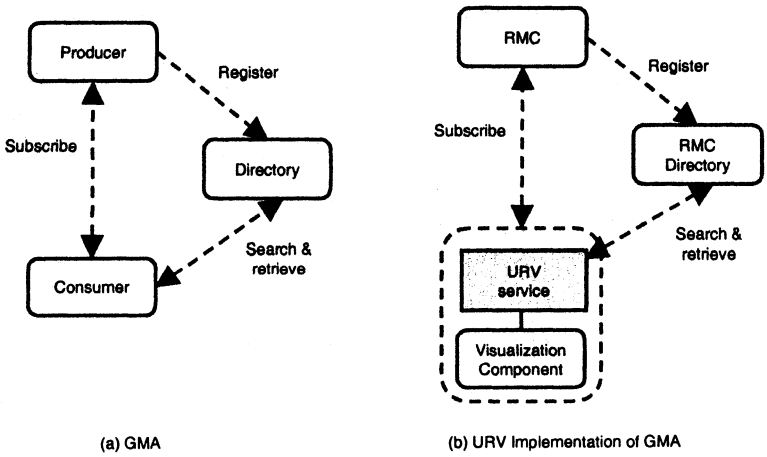


Figure 2. GMA, and URV Implementation with GMA.

3. URV services

As depicted in Figure 3(a), typical performance visualization environments are based on a one-to-one relationship between instrumentation and visualization. That is, there is often a single source or format of performance data prescribed for a particular visualization. Although there are exceptions, e.g., the model of Pablo [5], a user typically sees a graphical display that came packaged with the instrumentation. This is the case even with a many-to-one relationship, in which a basic graphical display type is paired with all performance data (as in Paradyn [4]). The availability of new performance data often means

that new visualizations will need to be programmed specifically for that tool environment. There is little chance that a visualization can be used with another tool, even though many of its attributes would satisfy the end-user needs. The main goal of URV is to extend the usability of visualization by providing

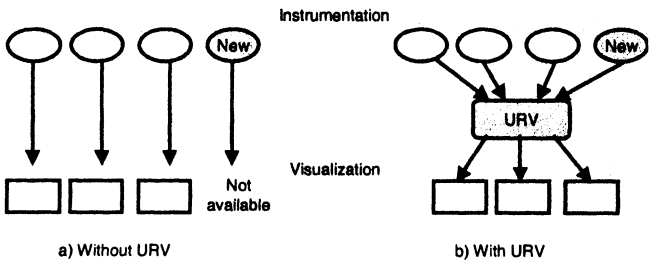


Figure 3. Performance Visualization With/Without URV.

a many-to-many relationship between instrumentation and visualization (e.g., between RMCs and VCs in URV). The many-to-many relationship supports the special cases of one-to-many and many-to-one relationships. In case of a one-to-many relationship, a single source of performance data can be viewed in several ways. This lets a user choose a perceptually effective visualization. On the other hand, a many-to-one matching enables a single visualization to be reused across several sources of performance data. For instance, a line chart can display CPU load over time, round-trip message time over time, etc. In addition, since performance views in URV are created in a uniform and consistent way, the views have syntactic and semantic compatibility with each other. A graphical representation in one performance view can be reused in a second view without losing its syntactic representation and semantic meaning. This feature enables the integration of performance views, whereby a new visualization is created by composing multiple visualizations.

The URV project has defined a set of services to support this many-to-many relationship, including relevant advanced software technologies in the areas of formal specifications, web services, and automatic visualization. URV services introduced in this paper address two key issues: connection between RMCs and VCs, and VC reuse.

3.1 Connection Technology & Services

A URV connection service provides communication and control between RMCs and VCs. Through a connection, components can transfer performance event data and status and control information. A connection in URV must meet the following requirements:

- Interoperability: A connection is not constrained to particular types or domains of components.
- Validity: A connection must establish a syntactically and semantically correct mapping between RMC and VC ports. In addition, validity should be based on efficacy of the VC, which requires knowledge about such aspects as data domain, effective information visualization, and user preferences. Thus, knowledge-based support is needed to fully validate an RMC-VC connection.
- Reliability: A connection must ensure the delivery of information between components.
- Transparency: A connection and the underlying connection process should be hidden from end-users.

Any RMC and VC can be connected as long as the connection is valid.

Heterogeneity is the most challenging issue in the design of connection services. The issue is two-sided: first, components may be heterogeneous; and second, the interaction may be heterogeneous. Independently developed software components may be written in different programming languages and tested on different platforms. Thus, the mechanics of connecting components is nontrivial. However, the dynamics of connecting components presents difficulties as well. Controlling the interaction between components, e.g., timing and synchronization, may depend on the context. For example, if latency is a concern, a connector cannot buffer data arbitrarily.

There are two approaches to address component heterogeneity: glue code and architecture definition. Glue code fixes mismatches between components that cannot be plugged together directly [31]. An example of glue code is a wrapper around a legacy application in order to use this application as a Common Object Request Broker Architecture (CORBA) [32] component. Glue code is often platform-dependent. A CORBA component only works within the same CORBA domain. Architecture definition adds another level of abstraction, moving differences between components to a lower level. An Architecture Description Language (ADL) [33] provides a formal description of the architectural structure of software systems. It allows the description of architectural styles, or families of systems, architectural instances, and component interaction. For example, Acme [34] is a simple, generic software ADL that can be used as a common interchange format; it provides systematic methods to convert architecture descriptions into real software components. Although ADL supports component connection at the architecture level, its practical use at the implementation level remains challenging.

One approach to address interaction heterogeneity is software channels or busses, which similar to ADL, adds a level of abstraction. ADL lets a designer

specify details about the communication protocol in an architecture description. For example, Polylith [41] uses software busses that declare various interaction attributes. Thus, a designer can select a bus that not only matches the component ports but also provides the communication and synchronization properties needed by the application.

The connection service in URV is applying the following technologies to deal with heterogeneity:

- Component heterogeneity: XML, Simple Object Access Protocol (SOAP) [20], and Web Service Definition Language (WSDL) [35]; and
- Interaction heterogeneity: Template connector based on software busses.

An XML-based remote method invocation stems from traditional remote service invocation and improves interoperability with messaging standards. SOAP is an example. SOAP does not address all aspects of component heterogeneity. It provides interfacing mechanisms, so that a VC can access services provided by a remote RMC. Actual description and identification of the services in a component will be addressed by a combination of SOAP and web services [30].

A component's context and interface, which is part its concept, can be described in WSDL. The rest of the component description is provided in a separate visualization descriptor. WSDL is based on XML. Without WSDL, all component meta-information is placed in a single descriptor. With WSDL, concept and context information can be separated into a WSDL entity, thus reducing the size of the component descriptor. In addition, WSDL includes constructs for representing service endpoints and messages. WSDL is supported by most SOAP toolkits.

More than a particular technology, the design of the connector must handle interaction heterogeneity. A URV connector mediates a connection between an RMC and a VC by buffering and synchronizing event data and by adjusting service invocation timing. Each connector is uniquely typed to provide specific interface characteristics. A template connector defines a class of connectors that inherit the interface characteristics and implement different interaction behaviors. Each template connector is configurable to support families of related components that share the same interface. Thus, connector design addresses a range of configurations and operations.

Figure 4 depicts the basic operations of a connection service and their sequencing. A connection service assists a user in identifying, locating, and using a visualization. Without such a service, a user either takes whatever visualization is packaged with a monitoring tool or spends time creating (or customizing) a visualization from scratch or off-the-shelf visualization software. The steps of the connection service may proceed automatically or with guidance from the user. First, the connection service identifies an RMC of interest with the

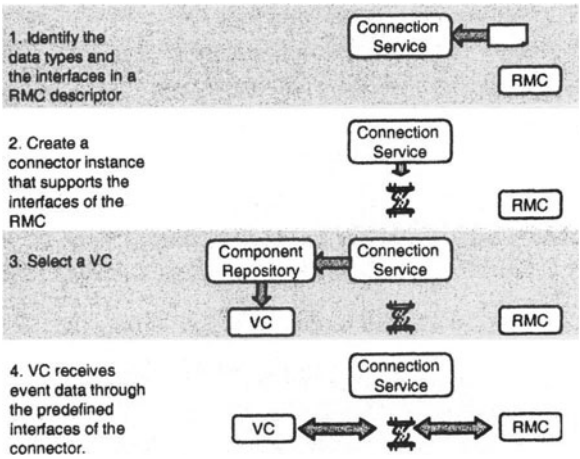


Figure 4. The Basic Operations of a Connection Service.

necessary contexts [Step 1]. Next, referencing the component's concept, the connection service creates a template connector that supports the interfaces of the RMC [Step 2]. Also the connection service identifies a VC having content that supports the RMC's concept [Step 3]. Then the template connector is configured; it is specialized by binding the names of services and by setting a control policy. Finally, the VC communicates with the RMC through the specialized connector [Step 4].

3.2 Reuse Technology & Services

A URV reuse service provides formal description of visual design knowledge and methods for sharable and reusable visualization components. Visual design knowledge in performance views consists of graphical representations (e.g., line, rectangle, mark), visual encodings (e.g., color, size, style), rules (e.g., how to position axes, how to position lines), and labels (e.g., title, x axis label). Reuse of visual design knowledge implies that, using the same knowledge, we are guaranteed to create multiple views that have the same graphical representations and the same perceptual and informative content. The formal description of visual design knowledge in URV must meet the following requirements:

- Versatility: A description should represent a group of multiple views based on syntactically compatible visual designs. Creating multiple instances from a single description reduces the number of descriptions that must be maintained.

- Extensibility: A description should allow the addition of new content.

- Hierarchy: A description must support a hierarchical representation of visual design knowledge.

Visual design knowledge is very diverse with rich semantic content, both of which make representation and reuse more challenging. A representation includes both graphical and stylistic content. Graphical representations are generally easier to define than stylistic qualities. That is because there is a finite set of accepted techniques and taxonomies for effective information visualization [9][12][13]. Graphical elements might be specified using a list. Style, however, is often dependent on a number of factors, including the domain, context and user. Rules may be needed to specify style. For example, a space-time diagram is commonly used to depict communication between resources. The graphical elements and their meanings are straightforward; e.g., a horizontal bar denotes the state of a resource over time by encoding the state as its color. However, even the organization of the graphical elements adds nontrivial content, such as: the horizontal bars are distributed along the vertical axis, one per resource, and lines are drawn between bars for each message, starting at the sender and ending at the receiver. The style of the visualization is even less precise, e.g., what should a visualization with hundreds of resources or messages look like? This diversity in visual design makes it hard to develop a consistent representation.

Two approaches are commonly used toward reusable visual design knowledge in the area of information visualization: taxonomy and automatic visualization. Researchers have constructed taxonomies of visualization techniques by examining the data domains that are compatible with the techniques [10][11][12]. The taxonomies help developers quickly identify various techniques that can be applied to the domains of interest. The taxonomy approach provides a basis to identify relevant visual design content, however, applying and implementing a technique is left to the developer. Automatic visualization methods [2][7] help developers create a view by suggesting an intermediate view based on predefined design knowledge at each stage of the visualization process. Tuning of the intermediate view eventually leads to a final view. This technique narrows the design space and has proven useful for non-visualization experts. However, in most systems based on this technique, a fixed set of data is visualized.

Reuse in URV takes several possible forms. Software wrapper technology is applicable for reusing visualization software modules that were not written as VCs. Most of the support for reusing visual design knowledge, as opposed to modules, needs to be developed, and the foundation of knowledge sharing in other domains must be leveraged. Reuse directions in URV are summarized below:

- Modify an existing visualization software module using a wrapper so that it qualifies as a VC and can be connected to. However, modules are often tightly coupled to an environment and not easily separated and wrapped.

- Create a high-level description of a VC with explicit representation of design knowledge. The high-level description is then synthesized to an implementation for a particular platform. For example, in chemistry, CML is a high-level description of chemical objects that is converted to Scalable Vector Graphics (SVG), which is a language for describing graphics in XML [36] [37].

- Develop directory services that locate VCs based on design content.

4. An Example URV scenario

Figure 5 shows one performance monitoring scenario for which URV is being developed. The scenario, which emphasizes multilevel monitoring, is based on a distributed collaborative virtual environment (CVE) being developed at Iowa State University [29]. In a CVE, potential users can build, manage, and execute complex collaborative worlds without in-depth knowledge of the underlying infrastructure. The CVE consists of several software technologies, including Plexus, VR Juggler, and Distributed Shared Object (DSO) middleware [28]. Plexus is a network data routing library that operates at the application level on top of the TCP/IP stack. It supports low-latency data networks for use within virtual environments. VR Juggler is a suite of APIs that abstract and simplify all interface aspects of VR programming, including display, virtual object manipulation, and graphic rendering. In addition, the DSO middleware provides consistent methods necessary for remote object creation, access, and navigation. Suppose a user develops a new VR application that requires a collaborative feature. Running the application within the CVE system, the user experiences an unexpected latency when rendering remote objects. The latency might be caused by a particular object rendering routine. In such a case, we pinpoint the problem by monitoring specific rendering processes. On the other hand, if the latency is associated with several consequences (e.g., an unnecessary retransmission caused by an inappropriate TCP buffer size results in poor remote object handling at DSO level), monitoring of a particular resource is not enough. Instead, multiple sub-analyses across levels should be conducted and integrated. For instance, a remote object access rate at DSO level, an object rendering (e.g., move, zoom) latency at VR Juggler level, and a retransmission rate at network level could be monitored. All performance views are synchronized over time and integrated into a single view. This integrated view helps identify correlation between object manipulation and network traffic.

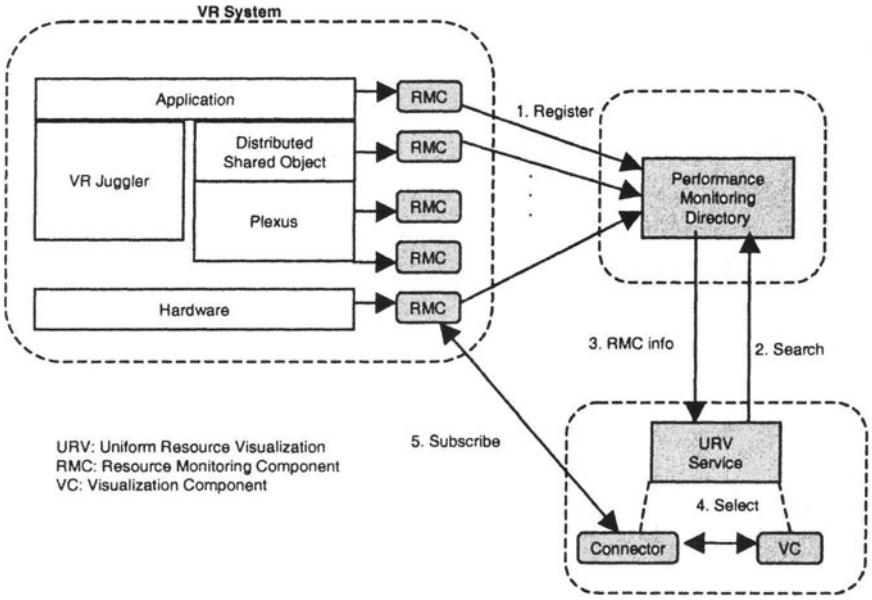


Figure 5. Multi-level Performance Monitoring of a VR System.

5. Related Work

URV is closely related to several works in the areas of grid resource monitoring, automatic visualization, and remote visualization.

In the area of grid resource monitoring, several monitoring frameworks have been developed to provide an extensible infrastructure for easy access of distributed performance events. Network Weather Service (NWS) [16] is an example of a framework to support persistent environments with data coming from measurement sensors. It aims to provide accurate forecasts of dynamically changing performance characteristics from a distributed set of metacomputing resources. The Information and Monitoring Services Architecture (a.k.a WP3) [23] is one of work packages initiated by the DataGrid project [39]. WP3 uses a relational model to represent performance events and HTTP servlet technology for interoperable communication. WP3 permits both job performance optimization as well as tracing. The Globus Heartbeat Monitor [26] provides a mechanism for monitoring the state of processes, including Globus system processes and application processes. It detects and reports the failure of processes. Several other monitoring tools, such as NetLogger [38] and Autopilot [14], are applicable with grid environments [27]. Compared with these frameworks,

URV also provides an open monitoring infrastructure, but this is not a main focus of URV. URV is not a performance monitoring tool per se. Instead, its focus is visualization. Current performance visualization techniques [8] [38] work well when analyzing certain performance problems, but often are insufficient when a problem domain expands.

In automatic visualization, a classic rule-based approach [6][9] constructs a visual presentation by using a set of predefined design rules that map the data to be conveyed onto desired visual formalisms. In this approach, a visualization design including the perspective and the graphical representations can be codified as sentences in a graphical language. A given set of data is broken down until it matches a primitive language. By composing each primitive language, a final language is constructed. Example-based design paradigms [7][18], start with a set of existing visual presentations (examples). Upon a user's request (e.g., finding presentations that are suitable for my data), this approach can search through its example database, and retrieve the most relevant examples. The retrieved examples can then be reused for or adapted to the new situation. URV takes advantages of both rule-based and example-based generations. Compared with existing techniques, URV's data domain differs significantly. While most automatic visualization techniques focus on visualization of static data of which types are known in advance, URV focuses on visualization of dynamic and heterogeneous data. In addition, while the current techniques are based on a limited set of views provided by a developer, URV is extensible to support new types of visualization defined even by end-users.

6. Summary

URV supports a novel approach to performance visualization. The visualization model for URV is based on two types of components, Resource Monitoring Component and Visualization Component, and connections between these components. An RMC dynamically connects to a VC, a process handled by a connection service. VC descriptions include visual design knowledge that is both reusable and composable. This paper has presented several challenges and approaches to component connection and reuse, and introduced corresponding URV services and technologies.

Acknowledgments

This work was supported in part by the National Science Foundation under grant no. ACI-0234252.

References

[1] Aleksandar M. Bakic, Matt W.Mutka,and Diane T.Rover. An On-Line Performance Visualization Technology. In *Proceedings of the IEEE Heterogeneous Computing Workshop*

in conjuction with the 2-nd Merged Symposium IPPS/SPDP 1999 - 13-th International Parallel Processing, pages 47-59, April 12-16 1999.

[2] R. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. Rivet: A flexible environment for computer system visualization. *Computer Graphics*, 34(1), February 2000.

[3] Ian Foster and Carl Kesselman (Editors). *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 1998.

[4] Barton P. Miller, Jonathan M. Cargille, R. Bruce Irvin, Krishna Kunchithapadam, mark D. Callaghan, Jeffrey K. Hollingsworth, Karen L. Karavanic, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37-46, November 1995.

[5] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference*, IEEE Computer Society, 1993.

[6] Rogowitz, B.E. and Treinish, L.A. An Architecture for Rule-Based Visualization. In *Proceedings of IEEE Visualization '93*, pages 236-243, 1993.

[7] Mei C. Chuah and Steven F. Roth. On the semantic of interactive visualizations. In *Proceedings of Information Visualization*, IEEE, San Francisco, pages 29-36, October 1996.

[8] Eric Shaffer, Shannon Whitmore, Benjamin Schaeffer, and Daniel A. Reed. Virtue: Immersive Performance Visualization of Parallel and Distributed Applications. *IEEE Computer*, pages 44-51, December 1999.

[9] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5, pages 110-141, 1986.

[10] S. K. Card and J. Mackinlay. The Structure of the Information Visualization Design Space. *IEEE Symposium on Information Visualization*, pages 92-99, 1997.

[11] Tweedie, L. Characterizing Interactive Externalizations. In *Proceedings ACM CHI'97*, pages 375-382, 1997.

[12] Hikmet Senay and Eve Ignatius. *Rules and principles in scientific data visualization*, Technical report, Department of Computer Science, George Washington University, Washington, D.C, 1990.

[13] Roth, S.F. and Mattis J. Data Characterization for Intelligent Graphics Presentation. In *Proceedings SIGCHI'90 Human Factors in Computing Systems*, pages 193-200, Seattle, WA, ACM, April 1990.

[14] R. Ribler, J. Vetter, H. Simitci, and D. Reed. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[15] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, and M. Thompson. A Monitoring Sensor Management System for Grid Environments. In *Proceedings of the IEEE High Performance Distributed Computing conference (HPDC-9)*, August 2000.

[16] R. Wolski, N. Spring,and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computing Systems*, 1999.

[17] Roth, S.F. and Mattis, J. Automating the Presentation of Information. In *Proceedings of the IEEE Conference on Artificial Intelligence Applications*, pages 90-97, Miami Beach, FL, February 1991.

[18] Michelle X. Zhou and Sheng Ma. Representing and Retrieving Visual Presentations for Example-Based Graphics Generation. *International Symposium on Smart Graphics*, Mar 2001.

[19] W. Smith, D. Gunter, and D. Quesnel. An XML-Based Protocol for Distributed Event Services. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1668-1674, 2001.

[20] D.Box, D. Ehnebuske, G. Kakivaya, A. layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*, The World Wide Web Consortium, 2000.

[21] B. Tierney, R. Aydt, D Gunter, W. Smith, M. Swany, V. Taylor, and R. Wolski. *A Grid Monitoring Architecture*, White paper, Global Grid Forum, 2002. Available on-line from http://www-didc.lbl.gov/GGF-PERF/GMA-WG/papers/GWD-GP-16-2.pdf.

[22] Zhou, M. and Ma, S. Representing and retrieving visual presentations for example-based graphics generation. In *Proceeding of Smart Graphics '01*, pages 87-94, 2001.

[23] WP3, Information and Monitoring Services, DataGrid Project, Available on-line from http://hepunx.rl.ac.uk/edg/wp3/.

[24] W. Tracz. Where does reuse start?. In *Proceeding of Realities of Reuse Workshop*, 1990.

[25] Kuk-Jin Lee and Diane T. Rover. A Component-based Framework for Uniform Resource Visualization," *Workshop on Software Visualization, International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada, May 2001.

[26] Globus Heartbeat Monitor Specification. http://www-fp.globus.org/hbm/heartbeat_spec.html.

[27] Z. Balaton, P. Kacsuk, N. Podhorszki and F. Vajda. *Comparison of Representative Grid Monitoring Tools*, Report of the Laboratory of Parallel and Distributed Systems, Computer and Automation Research Institute of the Hungarian Academy of Sciences, 2000.

[28] VR Juggler, http://www.vrjuggler.org.

[29] Hartling, P., Just, C., and Cruz-Neira, C. Distributed Virtual Reality Using Octopus. In *Proceedings of the IEEE Virtual Reality 2001 Conference*, Yokohama, Japan, March 2001.

[30] Web Service Architecture, http://www.w3.org/TR/ws-arch/.

[31] Jean-Guy Schneider. Components, Scripts, and Glue: A conceptual framework for software composition. Ph.D. thesis, University of Bern, Institute of Computer Science and Applied Mathematics, October 1999.

[32] CORBA, http://www.corba.org.

[33] D. Garlan, R.Allen, and J. Ockerbloom. Architectural Mismatch or, Why it's Hard to Build Systems out Existing Parts. In *Proceeding of the 17th Int. Conf. On Software Engineering*, April 1995.

[34] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, (3):213-248, July 1997.

[35] Web Service Description Language, http://www.w3.org/TR/wsdl.

[36] Henry S. Rzepa and Peter Murray-Rust. Chemical Rendering Using CML and SVG, http://www.ch.ic.ac.uk/svg/.

[37] Scalable Vector Graphics (SVG), http://www.w3.org/Graphics/SVG/Overview.htm8.

[38] Netlogger Visualization, http://www-didc.lbl.gov/NetLogger/nlv/nlvmain.html.

[39] The DataGrid Project, http://eu-datagrid.web.cern.ch/eu-datagrid/.

[40] Global Grid Forum, http://www.gridforum.org.

[41] J. M. Purtillo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151-174, January 1994.

A PERFORMANCE ANALYSIS TOOL FOR INTERACTIVE GRID APPLICATIONS *

Marian Bubak, Włodzimierz Funika
Institute of Computer Science AGH, al. Mickiewicza 30, 30-059 Krakow
ACC Cyfronet AGH, ul. Nawojki 11, 30-950 Krakow, Poland
{bubak,funika}@agh.edu.pl

Roland Wismüller
LRR-TUM, Technische Universität München, D-80290, Mńchen, Germany
and
Department of Software Science, University of Vienna, A-1090, Vienna, Austria
wismuell@in.tum.de, rw@par.univie.ac.at

Abstract

The paper presents the main features of a performance analysis tool for applications running on the Grid, which is not limited to standard measurements, but also comprises application-specific metrics and other high-level measurements. These requirements are not well addressed by the existing tools in the area of parallel and distributed programming. The paper outlines the main ideas as well as the design details of the G-PM tool developed within the EU CrossGrid project whose aim is to widen the use of Grid technology for interactive applications. The focus is on the operation of G-PM's components, its internal interfaces, as well as the graphical user interface.

Keywords: Grid computing, monitoring, performance analysis, measurement tools, interactive applications, performance visualization, instrumentation

1. Introduction

Nowadays grid computing has become one of the most efficient ways of solving data and compute intensive problems. In the EU Grid-oriented projects, Grid technologies are intended to enable a world wide, efficient processing and access to the huge amounts of data that will be produced by future high

*This work was partly funded by the European Commission, project IST-2001-32243, CrossGrid

performance computing experiments. A new EU project – CrossGrid[1] – has been started to extend existing Grid technologies by *interactive applications*. Besides providing the necessary Grid services and test-beds, four demonstrators for interactive Grid applications are developed in CrossGrid: simulation of vascular blood flow, flooding crisis support tools, data mining in High Energy Physics, and meteorology / air pollution simulation.

Applying Grid facilities to such large-scale interactive applications requires to exactly know the performance behavior of the applications. However, even with a good knowledge of the application's code, its detailed run-time behavior in a Grid environment is often hard to figure out, not the least because of the dynamic nature of this infrastructure. In order to support this task, a tool is being developed, named G-PM, which not only allows to measure just the standard performance metrics, but allows to determine higher-level performance properties and application specific metrics, like e.g. the response time and its breakdown.

To provide this kind of information, the G-PM tool will use three sources of data: performance measurement data related to the running application, measured performance data on the execution environment, and results of micro-benchmarks, providing reference values for the performance of the execution environment. This data is processed and transformed within G-PM in order to take into account the structural and architectural details of the Grid. In this way, information on performance properties and/or bottlenecks in applications can be provided without a need for the user to know the Grid's detailed architecture.

The paper starts with an overview on the requirements on performance analysis tools for interactive Grid applications. Based on it, the proposed functionality of the G-PM tool is outlined. Then we go to the design features of the tool focusing on the component structure, internal interfaces, and the user interface. In the end we sum up the discussion with presenting the current status of the work on the tool and the features of the first prototype of the tool.

2. Functionality of the G-PM tool for performance analysis of interactive applications

When analysing the demands of the performance analysis of interactive applications we can figure out the main requirements which should be met by a performance analysis tool. First, the tool should provide performance data meaningful in the application's context, including application-specific data. Examples of such data are "amount of disk I/O for a specific end-user's (application user's) interaction", "detailed breakdown of an interaction's response time", or "convergence rate of the numerical solver".

[1] See `www.eu-crossgrid.org` for more information.

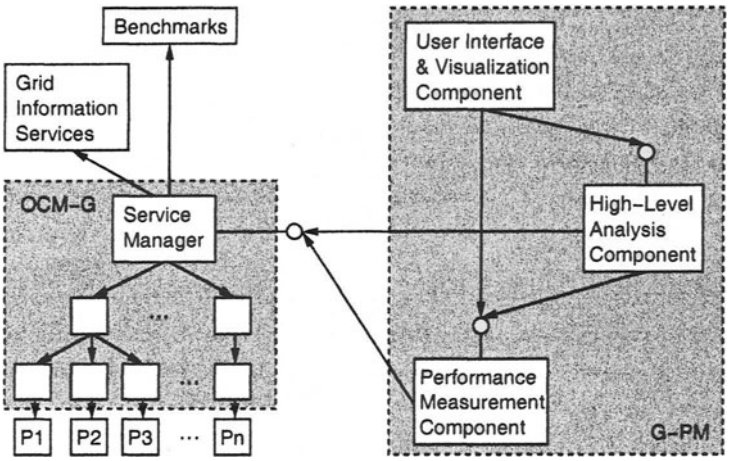


Figure 1. Structure of G-PM

Second, the performance analysis of an interactive application strongly demands to be carried out in on-line mode. This allows to correlate the performance data with the end-user's interaction patterns. Moreover, on-line data is a prerequisite for any kind of application steering, which is required e.g. by the CrossGrid flooding application.

Third, besides the application's performance data, the tool should at the same time being able to display data on the performance of the computing environment. This data can not only be used for steering an application, but also for the assessment of an application's performance.

Finally, the tool must well be integrated with the Grid infrastructure, especially the job submission services. Submitting a job with the performance analysis tool attached to it should be as simple as a normal job submission.

Meeting these requirements is a primary design goal of the G-PM performance analysis tool. The tool consists of three main components (see Fig. 1) that enable the user to optimize the performance of a Grid application:

1 a performance measurement component (PMC),

2 a component for high level analysis (HLAC),

3 a user interface and visualization component (UIVC).

The PMC provides the functionality for standard performance measurements of both Grid applications and the Grid environment. Among others, it allows to measure

- The resources utilization of an application, e.g. CPU and wall clock time, cache and memory accesses, disk accesses,
- The amount of data transfer in an application, e.g. messages exchanged between processes, data read from / written to files,
- Delays in the application due to communication and I/O operations,
- Resource information, like CPU load, CPU performance, link latency and bandwidth.

The results of these measurements can be directly visualized by the UIVC component. In addition, they can serve as an input to the HLAC component.

The ultimate goal of the HLAC is to provide application developers with more meaningful, application-specific performance data. This is achieved by providing a metrics specification language which allows the user to

- combine and/or correlate performance data from different sources. For instance, load imbalance can be measured by comparing the CPU time of a code region on each node. Likewise, the percentage of the maximum network bandwidth used by an application can be determined by comparing performance measurement data with benchmark data.
- measure application-specific performance metrics, like the time needed by one iteration of a solver, the response time of a specific request, the convergence rates of an iterative solver, etc.

We will present the HLAC in some more detail in subsection 3.3.

Finally, the UIVC allows the user to specify performance measurements and visualizes the performance data produced by PMC and/or HLAC. Details on the user interface are presented in subsection 3.4.

3. Design features of G-PM

3.1 Performance measurements vs. monitoring facilities

Fig. 1 shows that the G-PM tool is designed to be built on top of the OCM-G[3], a monitoring system compliant with OMIS 2.0 [7]. In order to start the discussion on the G-PM design, let us pay a bit of attention to the mode of operation of this monitoring system. The OCM-G is intended to provide the instrumentation of application processes as well as the basic measurement mechanisms like counters, timers and traces, and to handle all aspects related to the distributed nature of the target system. The monitoring system provides services, which either monitor events in the application, or perform actions like acquiring information on the application or manipulating processes or other objects, such as counters. The OCM-G is programmable in such a way that the

tool can specify actions to be executed whenever an expected event is detected. The programmability ensures that performance measurements can be evaluated in a distributed and thus efficient way.

In Fig. 1 the hierarchical structure of the OCM-G is shown, where the top-level Service Managers interface with other Grid information services in order to acquire infrastructure-related data. The Service Manager implements the universal monitoring interface OMIS [7], which not only supports performance analysis, but also other tools' functionality, e.g. debugging, by providing a wide range of comprehensive and environment-oriented services

The G-PM tool, which is implemented as a single process running on the user's local workstation. While the PMC component of the tool implements all standard metrics, HLAC operates on user-defined metrics. It uses the PMC to define sub-measurements based on standard metrics, but communicates directly with OCM-G in order to implement the monitoring of probes. The OCM-G is programmed such that when a probe event occurs, it either just notifies the HLAC, or directly reads and processes the sub-measurements. UIVC provides the user with windows to define measurements and to examine their results.

3.2 Internal and external interfaces of the G-PM

As can be seen from Fig 2 there are two major interfaces between these components:

- Measurement Interface: This interface allows to define performance measurements and to read their results. Both the HLAC and the PMC provide the same interface.
- OCM-G Interface: The interface to the monitoring system is based on OMIS.

As the PCM and the HLAC implement the same interface, the UIVC (and in turn also the user) can handle measurements based on user-defined metrics in the very same way as those based on standard metrics. The interface is implemented in C++, based on three (abstract) classes: `Metrics`, `MeasurementSpec`, and `ActiveMeasurement`. Below we shortly discuss the features of the classes.

A `Metrics` object defines a metrics, i.e. a property that can be measured, e.g. time, data volume, or number of times a particular event happened. The `Metrics` objects are organized in a hierarchical way that reflects a "refinement" relation. E.g. the metrics *time spent in communication* has subsidiary metrics like *time spent sending* and *time spent receiving*, which in turn have children like *time spent in MPI_Isend* etc.

A `MeasurementSpec` object contains the complete specification of a measurement that should be performed. It consists of

- the metrics to be measured,

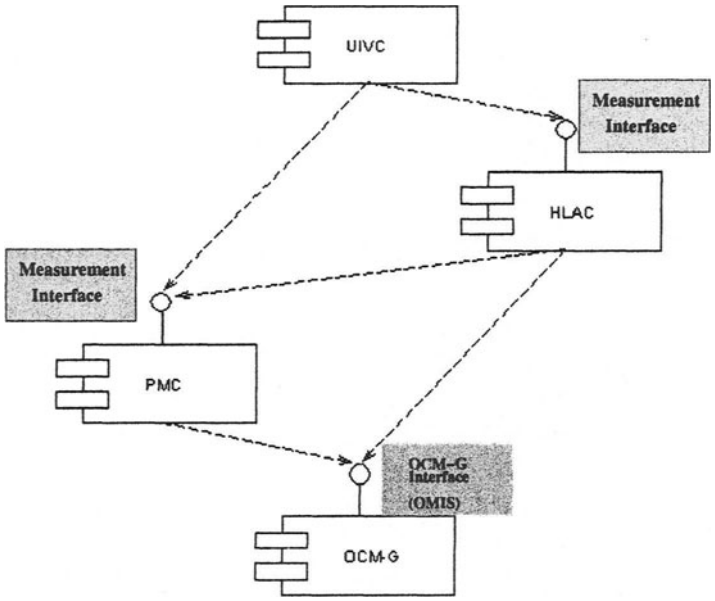


Figure 2. Components and interfaces of G-PM

- a set of application objects (processes, nodes, sites), where the measurement should be performed,
- a set of restrictions that limit the scope of the measurement,
- three flags specifying different measurement modes.

The scope of a measurement can be limited to certain functions inside the application. This allows, for instance, to measure the run-time of functions, or the volume of messages sent/received in selected functions. In addition, for metrics related to communication and I/O, the measurement can also be restricted to given partner objects (processes, nodes, sites, or files). The mode flags allow to select between

- a real time mode, where measurement data is plotted against wall clock time, and the virtual time mode, where the data is plotted against the number of executions of a specified probe,

- an integral mode, where the measurement result is the integral value since the measurement's start (e.g. the communication volume), and a normal mode, where the value just refers to the last measurement interval (resulting e.g. in the communication bandwidth),

- several summary modes, which determine whether for a set of application objects and/or restrictions the result should be a set of individual measurement values or just a single summary value.

A running measurement then is represented by an object of the `ActiveMeasurement` class. Among others, this class provides a method to read the measurement's results. By deriving concrete classes from the abstract `Metrics` and `ActiveMeasurement` classes, we are using different implementation strategies for different types of metrics, e.g. standard and user-defined ones.

3.3 User-defined metrics

In Sect. 2 we mentioned that the HLAC allows to measure application-specific metrics. It is clear that this cannot be done in a fully automatic way. Rather, the application itself must provide the tool with a minimal amount of truely application-specific information. In order to achieve this, the programmer must insert some instrumentation into the application's source code, in order to mark important events in the code, like e.g. the beginning and the end of a computation phase. The instrumentation consists of special function calls, named *probes*. A probe can receive any number of scalar parameters, which allows to pass application-specific data to the G-PM tool.

After the application is instrumented in such a way, the HLAC allows to measure three kinds of user-defined metrics: The first one is defined by an existing metrics (be it the standard metrics offered by the PMC, or another user-defined metrics), which is measured only during an execution phase delimited by any two probe executions. In addition, a metrics can also be defined by any parameter of any probe. Finally, a metrics can be derived from any existing set of metrics by aggregating or comparing their values.

```
IO_volume_for_interaction(Process[] processes, File[] files,
                          Region[] regions, TimeInterval currTime)
{
  volume[p][vt] = IO_volume(p, files, regions) AT end(p, vt)
                - IO_volume(p, files, regions) AT begin(p, vt);
  globalVol[vt] = SUM(volume[p][vt] WHERE p IN processes);
  result        = SUM(globalVol[vt] WHERE vt IN currTime);
  RETURN result;
}
```

Figure 3. Example of a user-defined metrics

An example of a metrics specification of the first kind is shown in Fig. 3. In this example, the programmer inserted two probes – *begin* and *end* – into the application, thus marking the beginning and end of an end-user interaction.

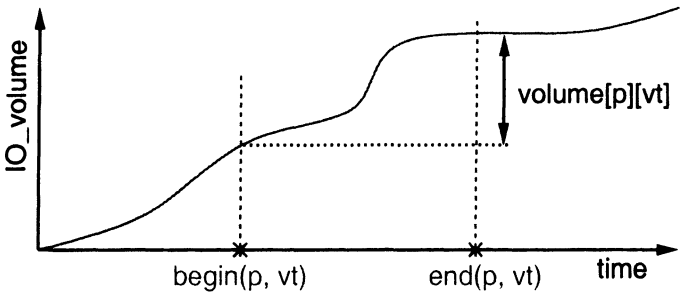


Figure 4. Idea of the metrics defined in Fig. 3

Based on these probes and the standard metrics `IO_volume` for the total volume of disk I/O, the user has defined a new metrics for the disk I/O during a single end-user interaction. For each process p, the value of this metrics is defined as the difference of `IO_volume` at the *end* event and `IO_volume` at the corresponding *begin* event, as visualized in Fig. 4. Note that `IO_volume` is the accumulated volume since the start of the measurement. The following two assignments in the metrics specification define how the result is accumulated over space (i.e. the processes) and time.

A major task of the HLAC is the optimized implementation of measurements based on user-defined metrics, such as the one shown in Fig. 3. Ideally, the processing of probe events should happen as locally as possible, i.e. in the best case within the context of the process where the event happens. This requires that the evaluation suggested by the metrics specification is performed in a distributed way. Some details about these problems and their solutions in the HLAC are presented in [12].

3.4 User interface

A graphical user interface provided by the UIVC consists of five different types of windows:

- a main window,
- a metrics definition window,
- a measurement definition window,
- a visualization definition window,
- a set of measurement visualization windows.

The main window represents the top level control panel for the tool. It shows a list of all active measurements and of all open visualizers. Using this window,

the user can manage (add, modify, or delete) measurements and visualization windows.

The metrics definition window allows the user to specify user-defined metrics: create a new one or edit the existing metrics. On the completion of the work, the metrics can be activated.

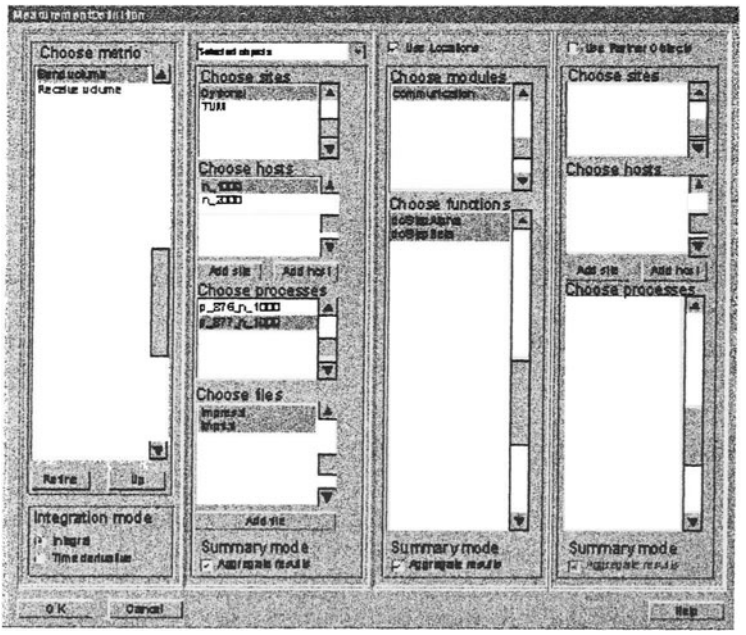


Figure 5. Measurement definition window

The measurement definition window (see Fig. 5) enables the user to define a new measurement or to change the parameters of the existing one. It creates and initializes a `MeasurementSpec` object. The window comprises four panels: The first one enables to select a metrics from the metrics hierarchy. The user can also choose the integration mode, specifying whether the result of a measurement is the currently measured value or the measured values integrated over time. In the second panel, we may choose whether the metrics should be measured for the whole application or for a subset of sites, hosts, processes, and files we are interested in at the moment. The third panel allows to narrow down a measurement to specific program regions, e.g. a set of modules or functions. The fourth "Partner object" panel is aimed at communication-type metrics, where measurements can be restricted according to source-target relations.

The visualization definition window enables to define an output window for a measurement. It allows to specify, how a selected set of performance

measurements should be visualized, e.g. via bar graphs or curves plotted against time. One of the two panels allows to select the type of display and the time mode of the measurement: real time or virtual time. The other panel allows to specify the partition of scale: linear or logarithmic, the behaviour of the display when the measured values are out of the scale, the initial boundary of the scale, and the update interval for the display.

The visualization windows' operation is dependent on the functionality of their displays. The bargraph window shows measured values in form of horizontal bars of variable length which specify the values of measurement. The multicurve window (see Fig. 6) displays multiple values simultaneously as scrollable curves.

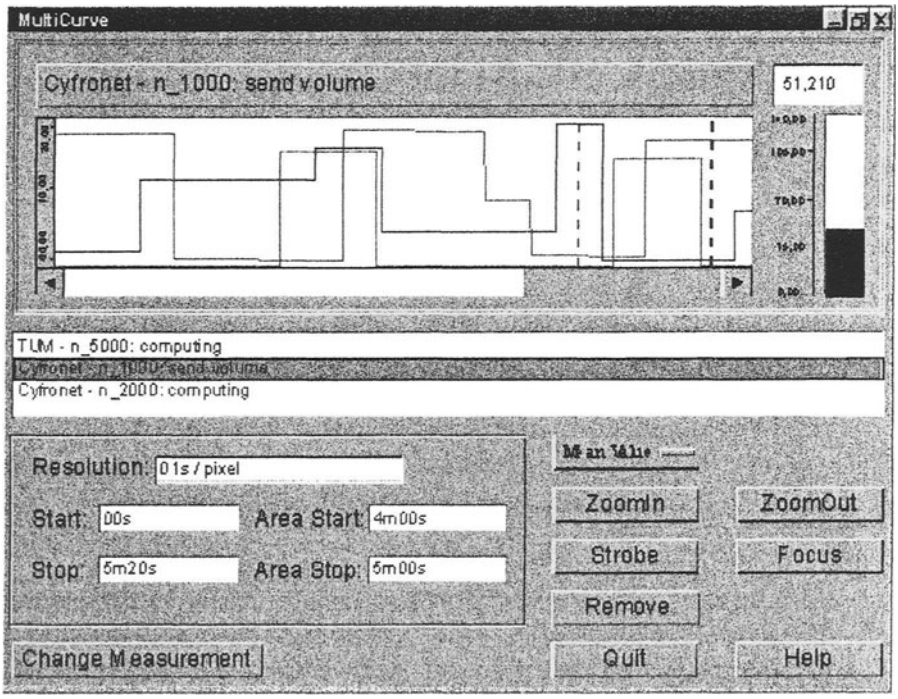


Figure 6. Multicurve plot

It enables to compute aggregated values from the measured ones, by selecting a time interval in the curve display and specifying the type of aggregate, e.g. mean, variance, or min/max value.

4. Related Work

By this moment there exist a number of performance tools,which are already adapted to the Grid [1]. The bulk of the tools which support the monitoring

of applications are based on an off-line analysis of event traces. But off-line tools are not helpful for interactive applications, since they cannot present the performance data concurrently with the end-user's interactions with the application. On the other hand, on-line tools for the Grid are available mainly for infrastructure monitoring, used for resource management. An example is the Network Weather Service [13] that forecasts network and node performance.

A notable exception is the monitoring system GRM [2], which is in its structure very similar to OCM-G. GRM is adapted to the Grid within the Datagrid project and offers on-line performance data of Grid applications. However, in contrast to G-PM, GRM is mainly based on event traces, does not allow for user-defined metrics, and does not provide infrastructure-related performance data.

A number of performance tools implement ideas closely related to the ones presented in this paper, but none of them implements all the ideas. Autopilot [11], a distributed performance measurement and resource control system, exploits a concept called *sensors* corresponding to our probes. User-defined instrumentation is used in the TAU performance analysis environment [8]. The SCALEA tool [10] supports application specific instrumentation via directives inserted into the source code. In all of these systems, however, the metrics connected with the user-defined instrumentation is already fixed by the instrumentation itself and cannot be (re)configured at run-time. The APART Working Group[2] has developed a specification language ASL [6] which allows to specify performance properties at a high level of abstraction. Although the ASL does not fulfill all the requirements on the metrics definition language needed by G-PM, many of the concepts outlined in Sect. 3.3 are based on the APART work.

Paradyn [9] is one of the few performance tools that strictly follow the on-line approach, where even the instrumentation is inserted and/or activated during run-time. Another such tool is PATOP [4], which is the predecessor of G-PM.

The main contribution of G-PM compared to existing tools is its unique combination of Grid awareness, on-line measurement, and automatic instrumentation for standard metrics on the one hand with a support for manual instrumentation and user-definable metrics on the other.

5. Status

The software design phase for the G-PM tool was recently finished. The first prototype of G-PM is going to be available at the beginning of 2003. This prototype will include some standard performance measurements, and some examples of higher-level metrics, but will not yet include fully user-definable

[2] See `http://www.fz-juelich.de/apart/`

metrics. The final version of G-PM will be ready by the end of 2004. Like all other system software developed in CrossGrid, G-PM will then be freely available via a public software license.

We expect that the highly pre-processed, application-specific performance information provided by G-PM will contribute to the support of program optimization and tuning, by simplifying the work of programmers and scientists on the Grid.

Acknowledgments

We would like to thank Mr. Tomasz Arodź and Marcin Kurdziel from AGH in Krakow as well as to Mr. Hamza Mehammed from TUM in Munich for their contribution. We are also very grateful to Mr. Bartosz Baliś, Tomasz Szepieniec, and Marcin Radecki for their remarks.

References

[1] Z. Balaton, P. Kacsuk, N. Podhorszki, and F. Vajda. Comparison of Representative Grid Monitoring Tools. Laboratory of Parallel and Distributed Systems (SZTAKI), LPDS-2/2000, 2000
ftp://ftp.lpds.sztaki.hu/pub/lpds/publications/reports/lpds-2-2000.pdf

[2] Z. Balaton, P. Kacsuk, N. Podhorszki,and F. Vajda. From Cluster Monitoring to Grid Monitoring Based on GRM. In: R. Sakellariou, J. Keane, J. Gurd, and L. Freeman (eds.), Euro-Par 2001 Parallel Processing, 7th International Euro-Par Conference, August 2001, Manchester, UK, pp. 874-881, vol. 2150, Lecture Notes in Computer Science,Springer-Verlag, 2001
http://link.springer.de/link/service/series/0558/papers/2150/21500874.pdf

[3] B. Baliś, M. Bubak, W. Funika, T. Szepieniec, and R. Wismüller. An Infrastructure for Grid Application Monitoring. In: Kranzlmüller, D. and Kacsuk, P. and Dongarra, J. and Volkert, J. (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, September - October 2002, Linz, Austria, 2474, Lecture Notes in Computer Science, 41-49, Springer-Verlag, 2002

[4] M. Bubak, W. Funika, B. Baliś, and R. Wismüller. On-Line OCM-Based Tool Support for Parallel Applications. In: Yuen Chung Kwong (ed.), Annual Review of Scalable Computing, vol. 3, ch. 2, pp. 32-62, World Scientific Publishing Co. and Singapore University Press, 2001
http://www.wspc.com.sg/books/compsci/4663.html

[5] M. Bubak and W. Funika and R. Wismüller. The CrossGrid Performance Analysis Tool for Interactive Grid Applications. In: Kranzlmüller, D. and Kacsuk, P. and Dongarra, J. and Volkert, J. (Eds.), Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, September - October 2002, Linz, Austria, 2474, Lecture Notes in Computer Science, 50-60, Springer-Verlag, 2002

[6] T. Fahringer, M. Gerndt, G. Riley, and J. L. Träff. *Knowledge Specification for Automatic Performance Analysis.* APART Technical Report, ESPRIT IV Working Group on Automatic Performance Analysis, November 1999.
http://www.fz-juelich.de/apart-1/reports/wp2-asl.ps.gz

[7] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 2.0). Shaker-Verlag, 1997, Aachen, Germany, vol. 9, ISBN 3-8265-3035-7
http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz

[8] A. Malony and S. Shende. Performance Technology for Complex Parallel and Distributed Systems. In: G. Kotsis and P. Kacsuk (eds.), Proc. Third Austrian-Hungarian Workshop on Distributed and Parallel Systems, DAPSYS 2000, 37-46, Kluwer, 2000
http://www.cs.uoregon.edu/research/paracomp/papers/dapsys2k.ps.gz

[9] B. P. Miller et al. The Paradyn Parallel Performance Measurement Tools. In: IEEE Computer, vol. 28(11): 37-46, Nov. 1995
http://www.cs.wisc.edu/paradyn/papers/overview.ps.gz

[10] H.-L. Truong and T. Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Programs. In: B. Monien and R. Feldman (eds.) Euro-Par 2002 Parallel Processing, 8th International Euro-Par Conference, August 2002, Paderborn, Germany, vol. 2400, pp. 75-85,Lecture Notes in Computer Science,Springer-Verlag
http://link.springer.de/link/service/series/0558/papers/2400/24000075.pdf

[11] J.S. Vetter and D.A. Reed. Real-time Monitoring, Adaptive Control and Interactive Steering of Computational Grids. In: The International Journal of High Performance Computing Applications, vol. 14, pp. 357-366, 2000

[12] R. Wismüller, M. Bubak, W. Funika, and B. Baliś. *A Performance Analysis Tool for Interactive Applications on the Grid.* Workshop on Clusters and Computational Grids for Scientific Computing, September 2002, Le Chateau de Faverges de la Tour, France. To appear.

[13] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. In: Future Generation Computer Systems, vol. 15, pp. 757-768, 1999

DYNAMIC INSTRUMENTATION FOR JAVA USING A VIRTUAL JVM

Kwok Yeung, Paul H.J. Kelly, Sarah Bennett
Department of Computing
Imperial College, London, UK
{kcy,p.kelly,s.bennett}@imperial.ac.uk

Abstract Dynamic instrumentation, meaning modification of an application's instructions at run-time in order to monitor its behaviour, is a very powerful foundation for a wide range of program manipulation tools. This paper concerns the problem of implementing dynamic instrumentation for a managed run-time environment such as a Java Virtual Machine (JVM). We present a flexible new approach based on a "virtual" JVM, which runs above a standard JVM but intercepts application control flow in order to allow it to be modified at run-time. Our Veneer Virtual JVM works by fragmenting each method's bytecode at specified points (such as basic blocks). The fragmentation process can include static analysis passes which associate dependence and liveness metadata with each block in order to facilitate run-time optimisation. We conclude with some preliminary performance results, and discuss further applications of the tool.

Keywords: Java, dynamic instrumentation, virtual machine, reflection, dynamic introspection, performance analysis

1. Introduction

Setting the Scene: optimizing Java RMI applications. The work we describe in this paper is part of a wider research programme at Imperial College, aimed at extending the technology of optimizing compilers to cross the boundary between systems on a network. Our main goal is to avoid unnecessary communications, and we have developed a prototype optimiser for Java applications that use Remote Method Invocation (RMI).

In looking for applications which might benefit from RMI optimization, we discovered the need for performance analysis tools. We also realised that the run-time optimisation framework which we had developed could also be used for performance instrumentation.

This paper presents the fruits of this insight.

The Veneer Virtual JVM. Static optimisation of Java is difficult due to dynamic binding, polymorphism, ubiquitous pointers, and dynamic class loading. Since communications (at least over a wide-area network) are expensive relative to computation, we can afford to invest in some run-time effort if it offers a reasonable prospect of reducing the number and/or size of messages required.

Although such optimisation could be done by extending a sophisticated Java Virtual Machine (JVM) such as the Jikes RVM [1], this would prevent users from using their chosen JVM, and would involve us in tracking JVM releases. Instead, we built Veneer, which operates on top of a standard JVM, running as a Java application. In effect, the framework is, itself, a JVM - which runs application class files in a controlled environment. Veneer is a "virtual" JVM, carefully designed to run reasonably fast by executing most of the application code directly — it jumps to the corresponding bytecode. It maintains control over execution by intercepting control flow; optional intercept points include method entry, basic blocks, and back edges.

This paper. The main contributions of this paper include:

- We give an overview of our Veneer Virtual JVM, an execution environment for Java bytecode applications which allows dynamic instrumentation, run-time optimization, and makes the results of static analyses available to inform run-time optimization.

- We briefly present JUDI, our Java Utility for Dynamic Instrumentation, a component-based environment which exploits Veneer's capability to modify an application on the fly.

- We briefly introduce the RMI optimisation tool which motivated this work.

We conclude with a discussion of directions for further work building on these ideas.

2. Related work

Dynamic instrumentation. The term "dynamic instrumentation" was coined by Hollingsworth and Miller to describe run-time patching of an application's binary code in order to monitor and measure the program's behaviour. Hollingsworth has published a portable library, DynInst [8], which supports this on a variety of processor types and operating systems.

The Paradyn Parallel Performance Tools [12] build on DynInst, to provide a tool for measuring and analyzing the performance of sequential, parallel, and distributed programs.

Dynamic instrumentation for Java. DynInst works by patching the application's instructions. Dynamic instrumentation for Java cannot be implemented this way, without exposing low-level implementation details of the JVM. There are a number of alternative approaches:

- Re-define the class using the Java Debug Interface (JDI) call `VirtualMachine.redefineClasses()`, introduced in Sun's JDK 1.4 [6]. This approach is used in ProbeMeister [13]. The overhead to do this is reported to be around 20 milliseconds for a small example, but increases with large classes since methods cannot be redefined individually, and JIT optimisation must be re-done. To reduce the overheads, Dmitriev [7] advocates refining the JDI with a call to redefine methods individually.
- Run the JVM in debugging mode, and set breakpoints to insert instrumentation. This is the approach taken by Popovici et al [14].
- Run the Java application in a virtual JVM. This is the approach used in our JUDI tool, and is presented in more detail in Section 4.1. We use the native JVM to execute application bytecode as much as possible, but have to intercept execution in order to retain control. The scheme suffers some overhead (see Section 5) on execution of all the application's code (apart from system libraries), but runs with JIT optimisation. Insertion and removal of instruments is very fast.

  Our vJVM was developed to provide a general framework for run-time optimisation, and is much more powerful than is needed for dynamic instrumentation. In Section 5 we explore some of the potential advantages of the using Veneer compared to the alternatives above.

Runtime Introspection and Optimization. The idea of interposing a software layer to monitor, intercept or optimise an application is interestingly ex-

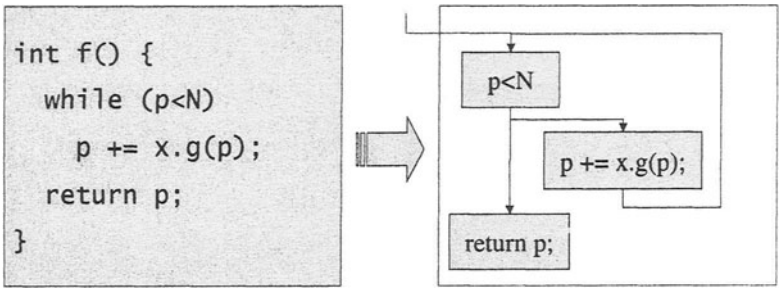


Figure 1. Fragmentation. When each class is loaded, each method may be fragmented according to a specified policy. For some purposes it is sufficient to intercept control flow only at method entry and exit. For our dynamic instrumentation tool we fragment at basic block boundaries - forks and joins in the method's control flow graph. Figure 4 shows the fragment graph for a real example.

plored by the Dynamo/Rio projects [2, 4]. Our virtual JVM provides essentially the same capability, and we encounter a similar problem of intercepting control flow efficiently.

Performance analysis for Distributed Java applications. Various tools already exist for analysing Java performance; for distributed applications, for example, the JaViz tool offers a useful solution [10]. Our goal for JUDI is to build an infrastructure for more ambitious instrumentation. We review some of the possibilities in Section 7.

3. The Veneer Virtual Java Virtual Machine

Our instrumentation and optimisation tools are built on a virtual Java Virtual Machine (vJVM), which is a JVM written in Java, running on a Java JVM. It runs most of the application code directly, by jumping to the corresponding bytecode, but selectively maintains control over execution by intercepting control flow. We can choose the level at which control flow is intercepted, e.g. at method entry, basic blocks, back edges.

The control flow is intercepted by "fragmenting" each method at class-load time. There are several different fragmentation policies: at basic block boundaries, at method entry/exit, at method calls, and at potential RMI call sites (used for our work on RMI optimisation, described in Section 6). Figure 1 shows a simple example of basic block fragmentation. The method body is split into blocks, an execution "plan", and the method entry is replaced by an "executor loop" that walks the control flow graph, invoking each block in turn. A method's control flow graph can be updated without synchronisation, as the application is running, allowing us to use this as a framework for dynamic instrumentation.

An architectural overview of the vJVM is given in Figure 2.

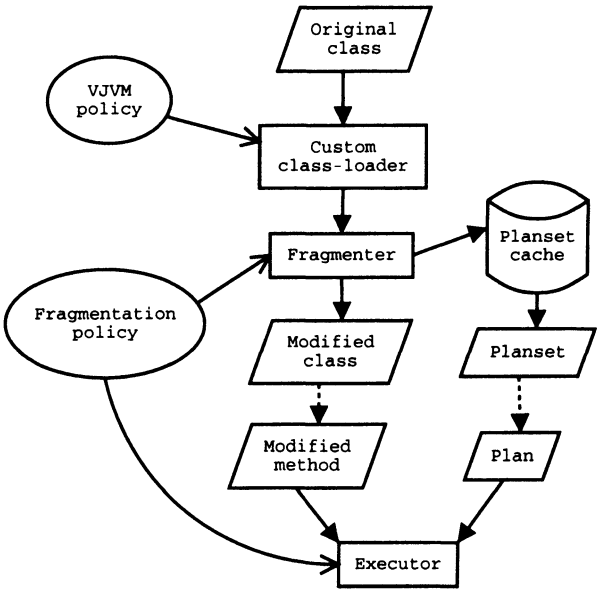


Figure 2. Architecture of the vJVM. We intercept the JVM's class loader, and replace each method of each class with a call to a plan executor. For each newly-encountered class we generate a planset, the fragmented code variants, which are traversed by the executor. Plansets are cached to avoid redundant work.

3.1 How it works

The Virtual Java Virtual Machine (vJVM) uses a custom class-loader to intercept classes as they are loaded. If the class belongs to the Java core library or to the vJVM, then its loading is delegated to the parent class-loader.

If modification is necessary, the methods of the class are processed one-by-one. The active policy is invited to generate *variants* for the method — the execution plans that can be executed in place of the original method body. Each method can have multiple variants, fragmented according to differing policies, although only one can be active at a time. The active variant may be changed at runtime, which effectively switches method implementations on-the-fly.

Finally, the modified class is loaded into the underlying JVM.

3.2 The fragmentation framework

The fragmentation framework breaks the body of a method up into blocks. Blocks represent sections of code from the original method, plus some additional meta-data. The structure of the original method is retained by building up a control-flow graph known as a *plan*, with the blocks forming the nodes of the plan. All the plans for a class are gathered up into a construct known as a *planset*.

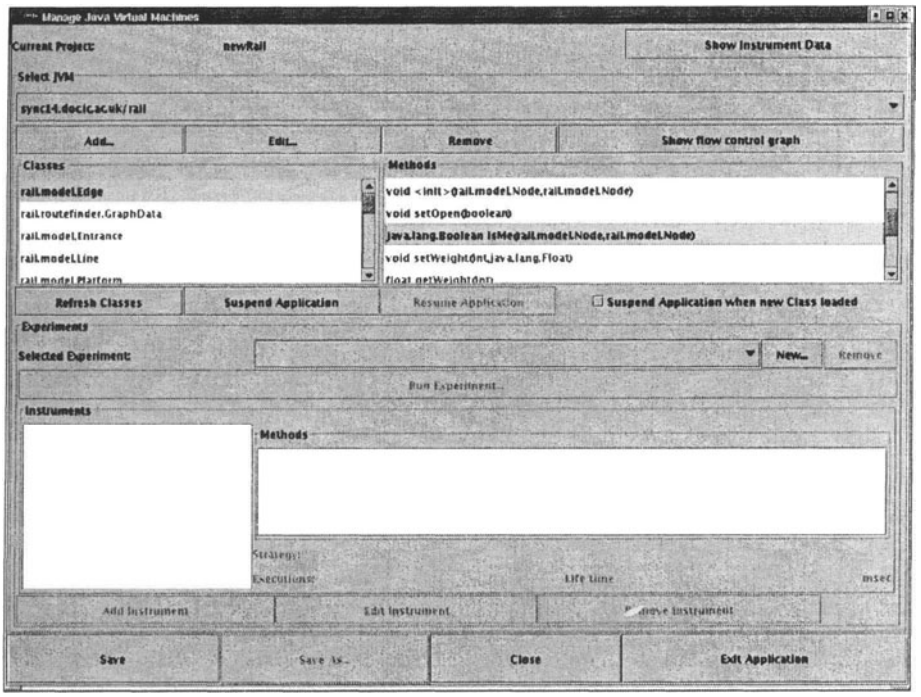


Figure 3. Java Utility for Dynamic Instrumentation. This prototype user interface connects to a specified, possibly remote, JVM. The user can then browse the classes and methods, and inspect their control-flow graphs. Instrumentation experiments are constructed, then deployed for a specified period, or until an instrument execution count is reached. Each instrument is packaged as an Instrument Strategy Component; an example is shown in Figure 4.

The method is executed by invoking a method in an *executor* class, which takes the plan as a parameter. The executor traverses the plan, executing the blocks as it visits them. The executor therefore takes on the role of an interpreter, and has full control over the way in which the blocks are executed.

Fragmentation policy. The way in which a method is fragmented and the executor that is used is determined by a user-defined policy. In general, we attempt to pack as much as possible within fragments, since these can be executed directly by the JVM and are therefore fast. Breaks between blocks are introduced where we need to regain control over the system, since at these points control is passed back to the executor. Parameterised blocks are used to identify certain types of instruction that may need to be treated specially by the executor.

Method entry. If a method is fragmented, its body is replaced by a sequence of instructions that initialises the locals array, retrieves the plan and executor for the current method, and then executes the plan using the executor.

Planset caching. The fragmentation process, which is implemented using the SOOT framework from Hendren's group at McGill University [18]) is rather involved, as data flow analysis is used to minimise fragmentation overheads (the metadata resulting from this analysis is used more aggressively in our RMI optimisation work). Plansets are cached in the local filesystem and reused if classname and SHA-1 signature match.

4. JUDI: Java Utility for Dynamic Instrumentation

JUDI is a prototype dynamic instrumentation tool for Java. It manages the deployment, removal, data collection and data visualisation for performance instrumentation experiments.

4.1 Instrumentation

Instrumentation is done by inserting instruments as additional blocks in the method plan. These instruments are then executed by the executor.

The instrument can access the method's parameters and locals, and call other classes, for example to trigger further instrumentation. For high-resolution timing, we used JNI to access a C/assembler routine which reads the Intel timestamp counter.

As illustrated in Figure 3, JUDI's client graphical user interface (GUI) connects to a set of remote vJVM's running fragmented code. The GUI allows the user to browse the remote systems' methods, and to upload instruments to the remote systems. The methods can be viewed by a visualiser (see Figure 4) which shows the plan as a graph (using OpenJGraph [9]). JUDI's record of accessible classes grows as classes are dynamically loaded, but persists from run-to-run to allow instrumentation of methods in advance of their execution.

4.2 Instrumentation Strategy Components

The unit of instrumentation deployment is an "Instrumentation Strategy Component" (ISC). This consists of:

- A set of **Instruments** - subclasses of a generic Instrument plan block. Instruments typically start, stop and log timers, or generate a log entry recording control flow, or data values (in aspect-oriented programming terms, this is the "advice").
- An **instrumentation strategy**. This is usually just whether the instrument is to be executed before, after, or before-and-after the specified

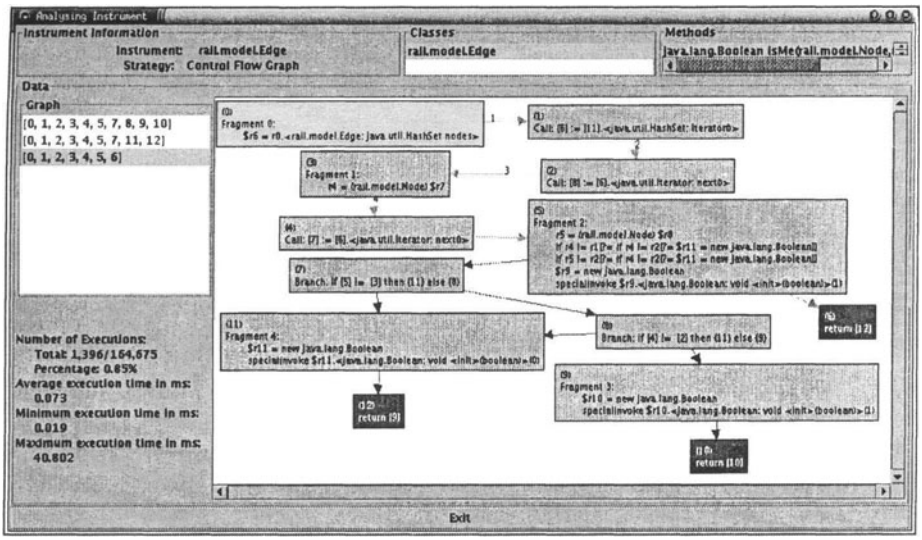


Figure 4. Fragmented execution plan for the example method `isMe`. In this example fragmentation has been applied at basic block boundaries, so that control flow within the method can be studied. The light-coloured block is the root of the plan which is always executed first. The small blocks are terminating blocks, where the method exits. The text in the boxes (unfortunately illegible here) shows disassembled code.

This display was generated by an Instrumentation Strategy Component (see Section 4.2) designed for control flow analysis. This ISC logs which control flow path was taken through the specified method, producing a histogram of path frequencies together with each path's mean, minimum and maximum execution time.

method, and whether it applies to the whole method, or every basic block in the method.

- **Instrumentation targets**: the set of program objects (methods, classes) to which the instrumentation strategy should be applied. If not the entire program, this is selected explicitly through the GUI.
- **Instrumentation data class**: instruments generate data, usually either a log or some kind of histogram.
- **Instrumentation analyser**: this is a GUI component for viewing the results from the experiment.

Figure 4 shows the results from an example ISC which traces control flow through selected methods. This ISC operates on methods fragmented at the basic-block level. The instrument is applied before each block, and simply logs the method and block id. The GUI allows us to select the particular method of interest, and view the results. The instrument analyser displays its control flow graph, the three distinct control flow paths which were taken through this

Table 1. Slowdown due to running benchmark applications under the Virtual JVM (without any instrumentation). The performance impact varies enormously. Note that system and standard library methods are not fragmented and run at full speed. Times are average of five runs.

Benchmark	*Exec^n time (JDK1.4)*	*Exec^n time (vJVM over JDK1.4)*	*Slowdown factor*
SpecJVM98_209_db	22.02s	24.30	1.10x
RouteFinder	3.73s	28.13	7.54x

method, the number of times each was executed, and the average execution time for each path.

5. Experimental results

We have evaluated JUDI using two substantial applications:

1 SpecJVM98_209_db (Data Management) Benchmark [16]. 1028 lines of code, 3 classes, 24 public methods.

2 RouteFinder, a railway route finder application written by a group of MSc students at Imperial College in spring 2002. 3192 lines of code, 17 classes, 145 public methods.

The experiments were run on a 1400MHz AMD Athlon processor with 512MB RAM, running Linux (Suse 7.2) using Sun JDK1.4.

Virtual JVM performance. Running an application under the virtual JVM leads to an inevitable increase in execution time due to the need to intercept control flow. The performance impact is given in Table 1. Our performance compares well with the reported slowdown of more than 700 for Sun's fully-interpretive JavaInJava virtual JVM [17], but is some way from matching the approaches discussed earlier based on the JDI `VirtualMachine.redefineClasses()` feature [13] and using breakpoints [14].

Veneer is at an early stage of its development, and we expect to improve its performance substantially. The implementation whose performance is reported here suffers a substantial overhead at each fragment boundary. The design allows for each method to have several different implementations, fragmented to different degrees. Thus, the overhead could be reduced to just a test or indirection on method entry (even this could be avoided in an implementation that can appropriate the method table). However, to ensure instrumented threads executed newly-inserted instruments promptly, the back-edges of loops generally also need to be intercepted.

```
void m(RemoteObject r, int a)
{
  int x = r.f(a);
  int y = r.g(a,x);
  int z = r.h(a,y);
  System.out.Println(z);
}
```

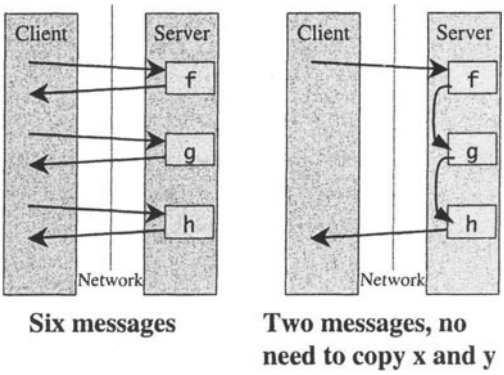


Figure 5. Aggregating adjacent, or near-adjacent RMI calls to the same server. Aggregation always reduces the number of messages. It may also reduce the amount of data transferred, since parameters used in multiple calls (such as `a` above) need be sent only once, and a result from one call passed as a parameter to another need not be routed via the client. Results which are not used by the client, such as `x` and `y`, need not be returned.

```
void m(RemoteObject r1,
       RemoteObject r2)
{
  Object a = r1.f(b);
  r2.g(a,b);
}
```

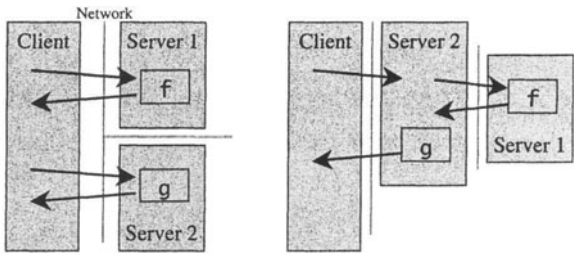


Figure 6. Aggregating adjacent, or near-adjacent RMI calls to different servers. If a result from one call (such as `a` above) is passed as a parameter to another, we can route the data server-to-server instead of server-to-client then client-to-server. This reduces marshalling costs and allows selection of a better server-to-server network path, if available. With a slow client-server connection, it may even be worthwhile purely to exploit common parameters such as `b`.

We could also improve performance by inlining instrumentation code into the modified method (or method variant) as it is loaded. This is likely to be particularly useful when large amounts of an application's code is to be instrumented.

6. Optimising RMI applications

Figures 5 and 6 illustrate the two main optimisations we have implemented to reduce communications in Java RMI applications.

To optimise RMI, we configure Veneer to fragment only methods which contain potential RMI call sites (interface invocations with `java.rmi.RemoteException` on the throw list). The fragmentation is used so that the run-time system is invoked before each potential RMI call. If the target object

is in fact remote, and the following fragment has no dependence on it, the RMI call is delayed. This way, RMI calls and local code are dynamically re-ordered. Eventually, when a dependence or externally-visible effect forces execution of the delayed RMI calls, the run-time system constructs an optimised execution plan which implements the aggregation and forwarding optimisations illustrated in Figures 5 and 6.

Performance results for RMI optimisation are currently being evaluated [19].

7. Conclusions and directions for further work

We have presented the Veneer virtual JVM. Veneer allows a Java application's behaviour to be selectively monitored and modified at run-time. Its design was motivated by our work on optimisation of RMI, where we combine static analysis with dynamic control flow. Veneer can also be used for performance instrumentation, and we presented JUDI, a prototype performance analysis tool. The tool is based on Instrumentation Strategy Components (ISCs), which combine instrumentation, an instrument deployment strategy, and instrument data analysis.

Given that much of this work is at an early stage, most of the interest lies in the directions for developing and applying it:

- **Veneer.** Veneer is at an early stage of development and we plan to improve its performance, and evaluate it more thoroughly. The tool has a number of interesting potential applications, particularly in security, which we are currently exploring.

- **JUDI.** The power of dynamic instrumentation lies in the ability to deploy instrumentation algorithmically. The idea is to formulate a hypothesis about a performance bottleneck, deploy an experiment to test it, then refine the hypothesis on the basis of the results. The approach has been explored in the Paradyn Performance Consultant [5, 15]. We have a preliminary implementation (as a JUDI ISC) which we are currently developing [3].

- **Aspects.** How should a JUDI user specify which program points a given instrument should be attached to? Aspect-oriented programming (AOP) languages such as AspectJ offer an answer to this [11]. AspectJ is based on a reflective model of a Java program, and uses this to define a language for specifying "joinpoints" (i.e. points at which code should be added or interposed). AspectJ supports wildcards on method type signatures, names and package pathnames. Wildcards on methods can be combined with some dynamic properties, such whether an object's run-time type matches a given pattern, or whether one method is called via another.

The PROSE tool of Popovici at al [14], which, as was mentioned earlier, implements dynamic instrumentation for Java by setting breakpoints via the debugging interface, is designed to support run-time deployment of aspects. They dynamically construct "crosscut" objects to represent how a specified code fragment is inserted into a running application.

We plan to redesign JUDI using an AOP-style language for characterising program points, and for classifying the measurements produced from instrumentation.

Acknowledgments

This work was supported by the EPSRC, through grant no. GR/R15566 (DESORMI). We would also like to thank Doug Brear, Thomas Petrou, Thibaut Weise and Tim Wiffen, who all contributed to the software development.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P.Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[3] Douglas J. Brear. JBolt: The Java bottleneck locator toolkit. Master's thesis, Department of Computing, Imperial College, London, UK, 2002.

[4] Derek Bruening, Evelyn Duesterwald, , and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[5] H. Cain, B. Wylie, and B. P. Miller. A callgraph based search strategy for automated performance diagnosis. In Arndt Bode, Thomas Ludwig II, Wolfgang Karl, and Roland Wismüller, editors, *Euro-Par*. Springer Verlag, LNCS 1900, 2000.

[6] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference, Tampa Bay, Florida, USA*, October 2001.

[7] M. Dmitriev. Application of the HotSwap technology to advanced profiling. In *First International Workshop on Unanticipated Software Evolution (USE2002)*, Malaga, Spain, June 2002. http://www.joint.org/use2002/sub/dmitriev-hotswapprof.pdf.

[8] Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In *Proceedings of SHPCC'94*, May 1994.

[9] Jesus M. Salvo Jr. Openjgraph - Java graph and graph drawing project, October 2002. http://openjgraph.sourceforge.net/.

[10] Iffat H. Kazi, Davis P. Jose, Badis Ben-Hamida, Christian J. Hescott, Chris Kwok, Joseph A. Konstan, David J. Lilja, and Pen-Chung Yew. JaViz: A client/server java profiling tool. *IBM Systems Journal*, 39(1):96–, 2000.

[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072, pages 327–355. Springer Verlag, LNCS 2072, 2001.

[12] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[13] Paul Pazandak and David Wells. ProbeMeister: Distributed runtime software instrumentation. In *First International Workshop on Unanticipated Software Evolution (USE2002)*, Malaga, Spain, June 2002. http://www.joint.org/use2002/sub/pazandak-ProbeMeister.pdf.

[14] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect oriented programming. In *1st International Conference on Aspect-Oriented Software Development (AOSD 2002), April 22-26*, Enschede, The Netherlands, 2002. http://ikplab11.inf.ethz.ch:9000/prose/webthings/aosd02.ps.

[15] P..C. Roth and B. P. Miller. DeepStart: A hybrid strategy for automated performance problem searches. In *EuroPar 2002*. Springer Verlag, 2002.

[16] Standard performance evaluation corporation (spec) jvm98 Suite, 1998. Available from http://www.spec.org.

[17] Antero Taivalsaari. Implementing a Java Virtual Machine in the Java programming language. Technical Report TR-98-64, Sun Labs, 1998. http://research.sun.com/kanban/JavaInJava.html.

[18] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of CASCON '99*, 1999.

[19] Kwok Cheung Yeung. *Automated Optimisation of Distributed Java Programs across Network Boundaries*. PhD thesis, Department of Computing, Imperial College, London, UK, 2002. In preparation.

AKSUM: A PERFORMANCE ANALYSIS TOOL FOR PARALLEL AND DISTRIBUTED APPLICATIONS *

Thomas Fahringer, Clovis Seragiotto, Jr.
Institute for Software Science
University of Vienna
Vienna, Austria
{tf,clovis}@par.univie.ac.at

Abstract Aksum is a multi-experiment performance analysis tool for message passing, shared memory and mixed parallelism programs; it automatically instruments the user's application, generates versions of this application using a set of user-supplied input parameters, collect the data generated by the instrumentation and analyzes it, relates the performance problems back to the source code, and compares the performance behavior across multiple experiments.

Aksum automatically searches for performance bottlenecks based on the concept of performance properties. In contrast to much existing work, performance properties are normalized (values between 0 for the best case and 1 for the worst case), enabling the user to interpret the resulting performance behavior. Aksum is highly customizable, which allows the user to build or define his own performance tool. Performance properties are defined in JavaPSL, and may be freely edited, removed from or added to Aksum in order to customize and speedup the search process. The performance properties found can be grouped, filtered, and displayed in several dimensions. Experiments with a material science code are shown in order to demonstrate the usefulness of our approach.

Keywords: automatic performance analysis, performance interpretation, performance specification language

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

1. Introduction

Developing applications that achieve high performance on parallel and distributed systems often requires multiple iterations of performance analysis and refinement. Moreover, for complex, data dependent applications, performance measurements from a specific execution based on a given problem and machine size may not be representative for other problem and machine size combinations.

Several tools have been developed to evaluate the performance of OpenMP [1], MPI [9], and HPF [2], as well as mixed programming paradigms such as OpenMP/MPI. Nevertheless, they typically cannot relate performance problems back to the source code, nor can they compare the generated performance data across several program executions for different machines, problem sizes, algorithms and so on. Most performance tools hard-code their instrumentation, search and analysis strategy, which prevent the user from customizing important aspects of a performance tool.

In this paper we introduce Aksum, which is a novel system for performance analysis that helps programmers to locate and to understand performance problems in message passing, shared memory and mixed parallel programs. The user must provide the set of problem and machine sizes for which performance analysis should be conducted. The search for performance problems (properties) is user-controllable by restricting the performance analysis to specific code regions, by creating new or customizing existing property specifications and property hierarchies, by indicating the maximum search time and maximum time a single experiment may take, by providing thresholds that define whether or not a property is critical, and by indicating conditions under which the search for properties stops. Aksum automatically selects and instruments code regions for collecting raw performance data based on which performance properties are computed. Heuristics are incorporated to prune the search for performance properties. We have implemented Aksum as a portable Java-based distributed system which displays all properties detected during the search process together with the code regions that cause them. A filtering mechanism allows the examination of properties at various levels of detail.

The remainder of this paper is organized as follows. The next section provides an overview of the architecture of Aksum. Section 3 briefly describes JavaPSL. Aksum's user portal is delineated in Section 4, followed by a description of the underlying search engine in Section 5 and the experiment engine in Section 6. Results for a material science application are presented in Section 7. Related work is discussed in Section 8. Finally, we conclude this paper with a summary and outline future work.

2. Aksum: Overview

Aksum has been designed to be a multi-experiment analysis tool, to a high degree independent of hardware and programming paradigm; it provides the user with a uniform and highly customizable interface to instrument an application, access and analyze performance data relative to several experiments, define how experiments are generated and executed, control the end of the search process, and define the search output. Once this info has been supplied (or the default values have been accepted), Aksum automatically conducts performance analysis without any user interference.

Figure 1 depicts Aksum's architecture. It relies on an *instrumentation system* to instrument the user's application and generate raw performance data. In the same way, Aksum assumes the existence of an *experiment management system* to generate experiments according to the user specifications, launch them and, if this is the case, transfer the data generated by the instrumentation to an *experiment data repository* (currently, we use SCALEA [12]and ZENTURIO [11]respectively as instrumentation and experiment management system). The *instrumentation engine* and the *experiment engine* link the external systems to *search engine*, Aksum's component which controls the entire search process. The output of the search process and user-supplied data that influence the search process flow to and from the *user portal*.

Aksum uses JavaPSL to define and customize performance problems in a systematic and portable way. JavaPSL, a Java-based specification language, allows the user to access the data in the experiment data repository and facilitates the definition of performance properties (classes that define a specific negative behavior in an application). Aksum has several pre-defined performance properties (such as inefficiency or load imbalance), stored in the *standard properties* repository, but the user may also define and store new properties in a *user-defined properties* repository. Instances of performance properties found in an application can be grouped, filtered, and displayed in several dimensions as well as plotted on charts.

3. JavaPSL and Performance Properties

JavaPSL [5]is a flexible API for describing experiment-related data (see Figure 2) and performance properties of applications by using syntax and semantic rules of the Java programming language; based on it, the user can define new performance properties without the need to understand the storage format of experiment data and without changing the implementation of the search tool that makes use of this API.

In JavaPSL, an *Application* is modeled as a set of *Versions*, each of them consisting of a set of *Source Files*. An *Experiment* refers to the execution of a sequential or parallel application version with a specific set of applica-

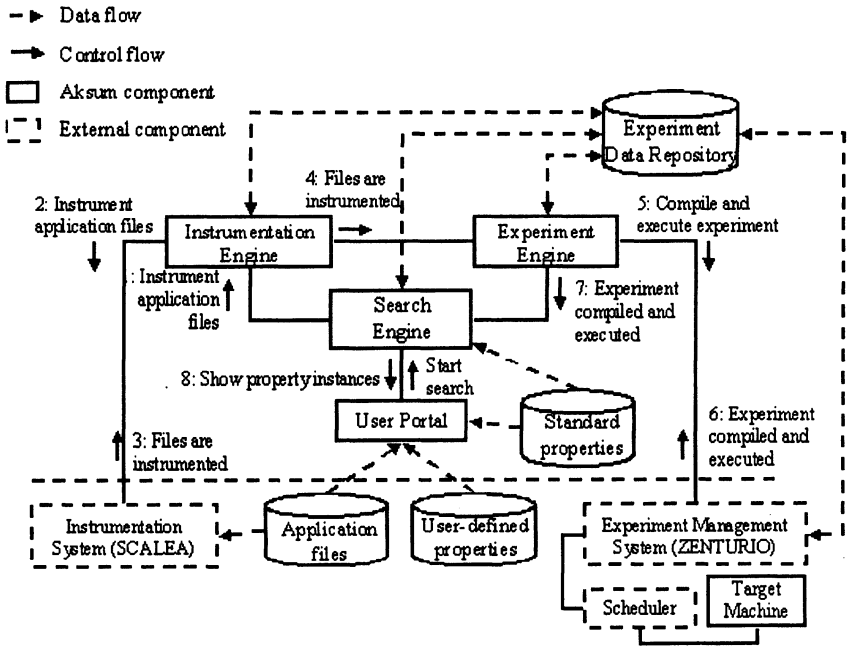


Figure 1. Aksum's architecture

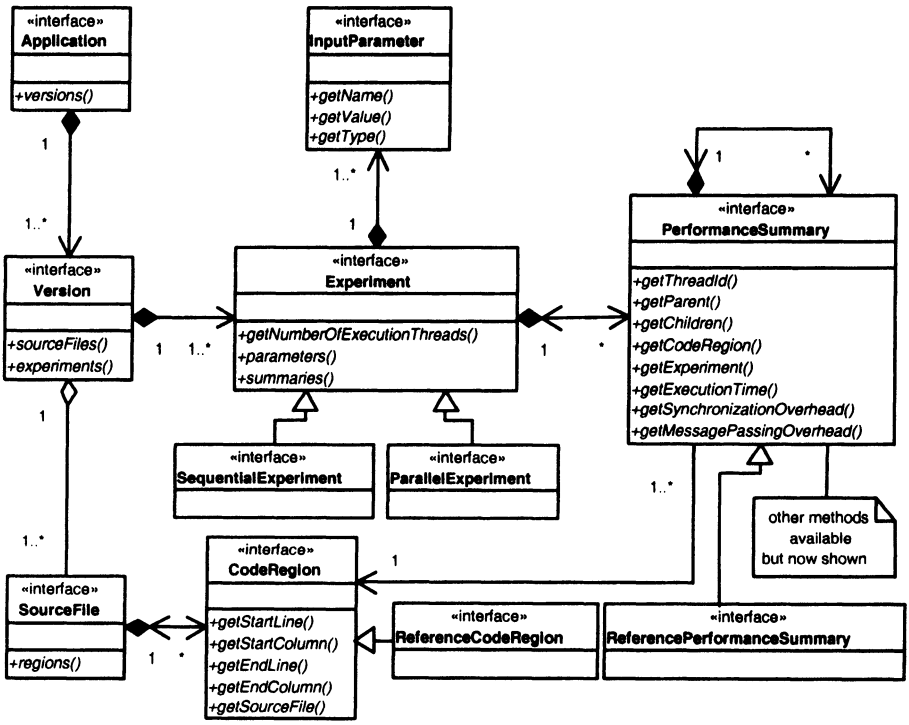


Figure 2. Experiment-related data represented by JavaPSL

```
public class SynchronizationOverhead implements Property {
   private float severity;
   public SynchronizationOverhead(PerformanceSummary summary,
                    ReferencePerformanceSummary refSummary) {
      severity =(float) summary.getSynchronizationOverhead()/
                     refSummary.getExecutionTime();
   }
   public boolean holds( )       { return severity > 0; }
   public float getSeverity( )   { return severity; }
   public float getConfidence( ) { return 1; }
}
```

Figure 3. SynchronizationOverhead defined in JavaPSL

tion *Input Parameters*. *Performance Summaries* describe information (e.g., execution and communication time, process/thread that executes the code region) about a specific execution of a (static) *Code Region* in a version's source code. *ParallelExperiment*, *SequentialExperiment*, *ReferenceDynamicCodeRegion*, and *ReferenceCodeRegion* are marker interfaces [8](that is, without any method declaration) used to indicate semantic attributes of classes. By using these interfaces, the property definition becomes clearer, more precise, and the property evaluation potentially faster.

A performance property (e.g. load imbalance, synchronization overhead) characterizes a specific negative performance behavior of a program. Every property must implement the interface *Property*, which contains three methods:

- *boolean holds*( ): returns true if the property instance holds. In this case the instance is called a true instance.
- *float getSeverity*( ): returns a value between 0 and 1 indicating how severe the property is (the closer to 1, the more severe the property is).
- *float getConfidence*( ): returns a value between 0 and 1 that indicates the degree of confidence in the correctness of the value returned by *holds*.

In the remainder of this paper, the term *property instance* is used when referring to an instance of a performance property.

Figure 3 shows a simple yet very common property definition example. The JavaPSL property *SynchronizationOverhead* is defined for a performance summary *summary* and a reference performance summary*refSummary* (e.g, the one referring to the main program or the enclosing subroutine). The severity is computed as the ratio between the corresponding synchronization overhead of *summary* and the execution time of *refSummary*. The property is true if the severity value is larger than zero.

```
public class SynchronizationOverheadAcrossSummaries
                    extends PropertySet implements Property {
   public SynchronizationOverheadAcrossSummaries(
                    ParallelExperiment parExp,
                    CodeRegion c,
                    ReferencePerformanceSummary refSummary) {
      PerformanceSummary.Iterator it = parExp.summaries(c);
      PerformanceSummary summary;
      while ((summary = it.next()) != null) {
          add(new SynchronizationOverhead(summary, refSummary);
      }
   }
   public boolean holds( )       { return anyHolds(); }
   public float getSeverity( )   { return getMaxSeverity(); }
   public float getConfidence( ) { return getMinConfidence(); }
}
```

Figure 4. SynchronizationOverheadAcrossSummaries defined in JavaPSL

Property instances may be grouped into *property sets* to create more complex properties. A property set defines, among others, the methods:

- *add*, which adds a property instance to the property set;
- *getMaxSeverity, getMinSeverity, getAvgSeverity*, which return respectively the maximum, minimum and average severity values across all properties already added to the property set;
- *allHold*, which returns true iff all of the properties added to the property set hold;
- *anyHolds*, which returns false iff none of the properties added to the property set holds.

For instance, based on the already defined property *SynchronizationOverhead*, one could define the property *SynchronizationOverheadAcrossSummaries* (see Figure 4), which is true if synchronization overhead is detected for any execution of a code region in a given experiment. Based on *SynchronizationOverheadAcrossSummaries*, it is possible to define *SynchronizationOverheadAcrossExperiments* (see Figure 5), true if synchronization overhead is detected in any execution of a code region in any experiment.

4. The User Portal

Aksum's user portal provides the user with a very flexible mechanism to control the search for performance bottlenecks, described in this section.

```
public class SynchronizationOverheadAcrossExperiments
                    extends PropertySet implements Property {
   public SynchronizationOverheadAcrossExperiments(
                    ParallelExperiment[] parExps,
                    CodeRegion c,
                    ReferencePerformanceSummary refSummary) {
      for(int i = 0; i < parExps.length; i++) {
          add(new SynchronizationOverheadAcrossSummaries(
                  parExps[i], c, refSummary);
      }
   }
   public boolean holds( )       { return anyHolds(); }
   public float getSeverity( )   { return getMaxSeverity(); }
   public float getConfidence( ) { return getMinConfidence(); }
}
```

Figure 5. SynchronizationOverheadAcrossExperiments defined in JavaPSL

4.1 Property Hierarchy

Properties are hierarchically organized into tree structures called *property hierarchies*, which are used to tune and prune the search for performance properties. For example, one may assume that, if an application is efficient, there is no need to compute its load imbalance. This assumption can be encoded in a specific property hierarchy by placing the property *LoadImbalance* under the property *Inefficiency*. Another example would be the definition of a property hierarchy without any communication properties if it is known that the application is encoded as an OpenMP code and runs on a shared memory machine.

Each node in the property hierarchy represents a performance property and is described by three elements:

- *Performance property name*: the name of the performance property associated with this node; the property definition is stored in a property repository (defined by the user or provided by Aksum).
- *Threshold*: a value that is compared against the severity value of each instance of the property represented by this node; if the severity value is greater than or equal to this value, then the property instance is *critical* and will be included in the list of *critical* properties.
- *Reference code region*: the type of the reference code region (currently main program, enclosing outermost loop or subroutine) that will be used to compute the severity of instances of the property represented by this node (the property must have declared in its constructor that it needs a *ReferenceCodeRegion* or a *ReferencePerformanceSummary*).

There are three standard property hierarchies provided by Aksum, covering message passing, shared memory and mixed parallel programs (the user can define and store new property hierarchies from scratch or based on these predefined hierarchies). The reference code region for every property node in the predefined property hierarchies is per default set to the main program.

4.2 Application Files, Command Lines and Directories

An application consists of various files – denoted *application files* in the remainder of this paper – which are divided into *instrumentable* and *non-instrumentable* files. Instrumentable files are source codes that must be instrumented for performance metrics (overheads and timing information) whereas non-instrumentable files refer to source codes the user does not want to be instrumented and any other files necessary to execute an application (e.g., makefiles, scripts, and input files). For instrumentable files, the user can also define the code regions that should be instrumented. If not specified, then Aksum assumes that the entire file must be instrumented. The user must also provide the compilation and execution command lines and associated directories for a selected application. This information is later used by the experiment engine.

4.3 Application Input Parameters

An application input parameter defines a string that should be replaced in some or all of the application files and in the execution and compilation command lines before the application is compiled and executed. Application input parameters are defined by the quintuplet (*name*, *search string*, *value lists*, *file set*, *type*), where *name* is a unique name that identifies the parameter, *search string* represents the string to be substituted, *value list* denotes the list of values the search string will be replaced with, *file set* describes the set of application files in which the search string will be searched and replaced, and *type* indicates if the parameter is machine or problem size related (or neither of them). If, for all input parameters, every *search string* is replaced in the associated *file set* with one of the values in the *value lists*, the resulting set of files is called an *application instance*. The generation, compilation and execution of an application instance is called an *experiment*. Figure 6 shows how the user provides values for the input parameters under the user portal. Two additional options ("Line by line" versus "Cartesian product") affect the number of experiments generated and the parameter value combinations for each experiment; they are detailed in Section 6.

4.4 Checkpoints

Aksum supports checkpoints to stop the search for properties. A checkpoint is defined as follows:

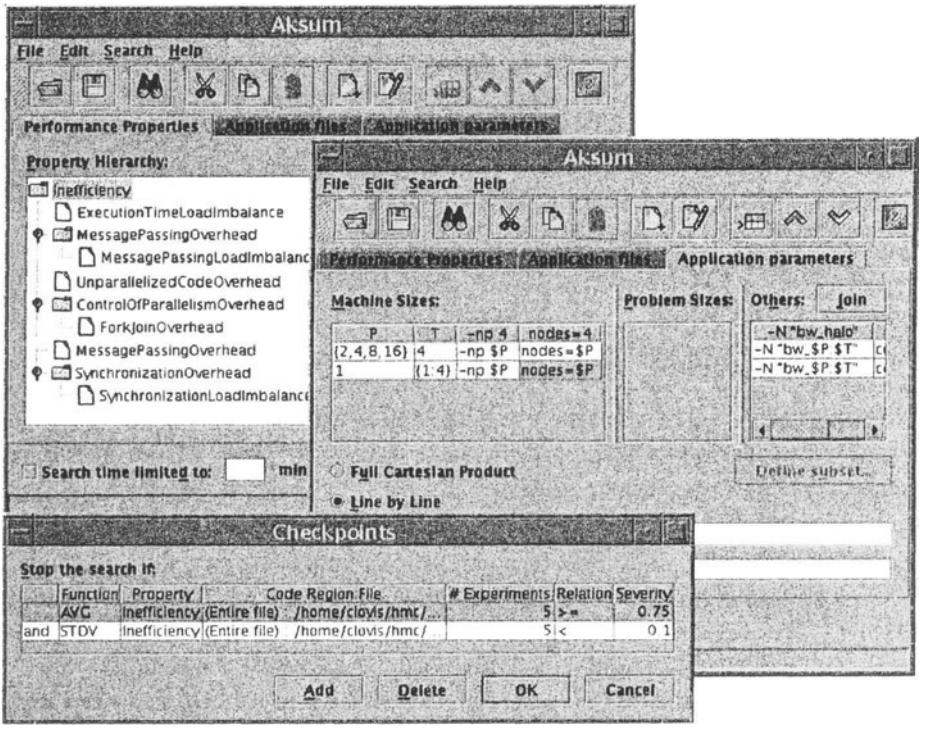


Figure 6. Aksum's user portal

op(severity(property, code region, number of experiments)) *relop* value where *op* $\in$ {maximum, minimum, average, standard deviation} and *relop* $\in \{>, \geq, <, \leq, =, \neq\}$.

Figure 6 shows a checkpoint definition that stops the search process if the average inefficiency for the entire program in the last 5 experiments is above 0.75 with standard deviation less than or equal to 0.1.

5. The Search Engine

The search engine controls the search for performance properties based on the experiments conducted. The search engine:

- decides which code regions should be instrumented and the performance overheads and timings that should be determined for these regions, and
- controls the evaluation of properties based on the property hierarchy.

Currently, the search engine issues requests for instrumenting all subroutines, outermost loops, subroutine calls (in particular, calls to the MPI library), and OpenMP constructs within the file regions of interest. After instrumentation, an application instance can be accessed by the experiment engine in order to be compiled and executed on the target machine. Every time an experiment terminates normally, the resulting performance data is stored in the experiment repository and the search engine is notified about that. Thereafter, the search engine evaluates the property hierarchy as follows:

- It tries to instantiate every property in the property hierarchy with the data (represented in JavaPSL) stored in the experiment repository.
- All true property instances are examined against the threshold for the given property in the property hierarchy. If the severity value of the property instance is above the threshold, the instance is in the list of *critical property instances*, which is the output of the search engine.

The search engine traverses the property hierarchy in a depth-first order; it prunes the evaluation of the property hierarchy and avoids the creation of false and non-critical property instances by using a well-defined heuristic. But, before showing it, we need to describe some important notation. A *static code region* (or simply code region) is a non-empty set of program statements. If a code region q may be executed by a thread t in a process p, we call a *child* of q any code region that may be executed by thread t in process p after the execution of q has started but not yet ended. Any code region that has q as a child is a *parent* of q. Moreover, if q' is a child of q, and there is no code region q'' that is a parent of q' and child of q, then q' is an *immediate child* of q, and q is the *immediate parent* of q'.

Now, let q be a code region, e an experiment, and DC(q, e) = { (p, t, q, y) | t is a thread in process p that executes q in experiment e, and y is one immediate parent of q that was being executed by thread t and process p when the execution of q started}. Every element d = (p, t, q, y) of DC(q, e) is called a *dynamic code region.* We will denote y as *parent(d)*, e as *experiment(d)*, and q as codeRegion(d). There is a one-to-one correspondence between dynamic code regions and the performance summaries defined in JavaPSL (an instance of a *PerformanceSummary* may be seen as a dynamic code region with performance information associated).

A *containment relation* between (dynamic) code regions needs also to be defined:

- Given two dynamic code regions, d and d', then d contains d' if $d = d'$ or there is a sequence of dynamic code regions $(d_1, \ldots, d_n)$ such that $d_1 = d$, d_n =d', and d_i = parent(d_{i+1}) $\forall i, 1 \leq i < n$.

- Given a dynamic code region d, a static code region q, and an experiment e, then q contains d in experiment e if there is a dynamic code region d' such that experiment(d') = e, codeRegion(d') = q, and d' contains d.

Figure 7 gives an example of code regions, dynamic code regions and the containment relation among them.

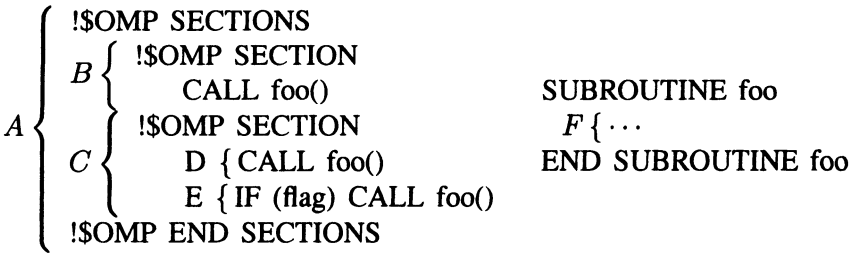
Based on the definitions above, we can define the heuristics used by Aksum:

- Given a property P with constructor P(..., PerformanceSummary ps, ...) and a dynamic code region d, if a non-critical instance of P is created with $ps = d$, then Aksum will try to create neither instances of P with a dynamic code region that d contains nor instances of a property under P in the property hierarchy with d or a dynamic code region that d contains.

- Given a property P with constructor P(..., Experiment exp, ..., CodeRegion cr, ...), an experiment e and a code region c, if a non-critical instance of P is created with $exp = e$ and $cr = c$, then Aksum will not use e and c to create any property Q under P in the property hierarchy with constructor Q(..., Experiment exp, ..., CodeRegion cr, ...), nor will Aksum use a dyanmic code region that c contains in experiment e to instantiate a property R under P with constructor R(..., PerformanceSummary ps, ...).

6. The Experiment Engine

The experiment engine is responsible for translating the input parameters given by the user to a form the Experiment Manager System can understand in order to create and launch application instances. Based on the description of a generic application input parameter v (see Section 4.3), which is a quin-



- Code regions: A, B, C, D, E, F
- Two experiments executed: $e1$ (with flag = true, thread $t1$ in process $p1$ executed code region B, thread $t2$ in process $p1$ executed code region C) and $e2$ (with flag = false, thread $t3$ in process $p2$ executed code region B, thread $t4$ in process $p2$ executed code region C)
- Fourteen dynamic code regions (performance summaries):
  - in experiment $e1$:
    $a = (t1, p1, A, -)$, with the dash meaning "no parent"
    $b = (t1, p1, B, a)$
    $c = (t2, p1, C, a)$
    $d = (t2, p1, D, c)$
    $e = (t2, p1, E, c)$
    $f_1 = (t1, p1, F, b)$
    $f_2 = (t2, p1, F, d)$
    $f_3 = (t2, p1, F, e)$
  - in experiment $e2$:
    $a' = (t3, p2, A, -)$
    $b' = (t3, p2, B, a')$
    $c' = (t4, p2, C, a')$
    $d' = (t4, p2, D, c')$
    $f_1' = (t3, p2, F, b')$
    $f_2' = (t4, p2, F, d')$

$a \supseteq b \supseteq f_1, a \supseteq c \supseteq d \supseteq f_2, c \supseteq e \supseteq f_3, a' \supseteq b' \supseteq f_1', a' \supseteq c' \supseteq d' \supseteq f_2'$

For Experiment $e1$, we say:
$A \supset a, b, c, d, e, f_1, f_2, f_3$, $B \supset b, f_1$, $C \supset c, d, e, f_2, f_3$, $D \supset d, f_2$, $E \supset e, f_3$, $F \supset f_1, f_2, f_3$
and for Experiment $e2$:
$A \supset a', b', c', d', f_1', f_2'$, $B \supset b', f_1'$, $C \supset c', d', f_2'$, $D \supset d', f_2'$, $F \supset f_1', f_2'$

Figure 7. Code regions and dynamic code regions

tuple (*name*(v), *searchString*(v), *valueList*(v), *fileSet*(v), *type*(v)), we define an application instance as follows:

Let $(v_1, \ldots, v_n)$ be the list of application input parameters. The set of instrumented application files is denoted an *application instance* AppInst$(s_{v_1}, \ldots, s_{v_n})$ iff $\forall i$, $1 \leq i \leq n$, the string *searchString*(v_i) has been substituted by a defined string s^{v_i} of *valueList*(v_i) in every file of *fileSet*(v_i)[1].

The experiment engine requires that experiments be generated according to the following policies:

- Let $(v_1, \ldots, v_n)$ be the list of input parameters, and $(s_1, \ldots, s_n)$ elements in *valueList*(v_1), ..., *valueList*(v_n), respectively. Recalling (Section 4.3) that the user has two options to control the generation of application instances, "cartesian product" or "line by line", AppInst$(s_{v_1}, \ldots, s_{v_n})$ is created for every different value combination of (specified) values in the value lists if the cartesian product option was chosen, while, for the option line by line, AppInst$(s_{v_1}, \ldots, s_{v_n})$ is created iff position(s_{v_i}) = position(s_{v_j}) $\forall i, j : 1 \leq i, j \leq n$, where position$(s_{v_i})$ denotes the position of an element s_{v_i} in *valueList*(v_i).

- Let w_1 = AppInst$(s_{v_1}, \ldots, s_{v_n})$ and w_2 = AppInst$(s'_{v_1}, \ldots, s'_{v_n})$. Then w_1 is generated (and later on executed) before w_2 iff $\exists\, k$ such that $1 \leq k \leq n$ and position$(s_{v_i}) \leq$ position(s_{v_i}) $\forall i, 1 \leq i \leq k$. This option enables the specification of an order for the generation and execution of application instances, which can be important when defining checkpoints.

For instance, given a parameter (p, "-mp 2", ("-mp = 1", "-mp = 4"), myScript.sh, machine-size related) and a parameter (q, "THREADS = 1", ("THREADS = 2", "THREADS = 4"), myScript.sh, machine-size related), the option "line by line" will cause the creation of two application instances, the first with the strings "-mp 2" and "THREADS = 1" substituted in the file myScript.sh respectively by "-mp 1" and "THREADS = 2", and the second with them replaced with "-mp 4" and "THREADS = 4". On the other hand, the option "cartesian product" would create four applications instances, the first with the replacements "-mp 1" and "THREADS = 2", the second with "-mp 1" and "THREADS = 4", the third with "-mp 4" and "THREADS = 2", and the fourth with "-mp 4" and "THREADS = 4".

For convenience, a value between braces has a special meaning in a value list and denotes a set of values. For instance, the value list ("-mp = {1, 2, 4, 6, 8}") is equivalent to ("-mp = 1", "-mp = 2", "-mp = 4", "-mp = 6", "-mp = 8"). In particular, the string "*a*:*b*:*c*" inside braces, where a, b and c are real numbers, is a special shortcut for "a_0,..., a_z", where $z = \lfloor (b - a) \div c \rfloor$ and

[1] Value lists may contain undefined (i.e., non-specified) strings.

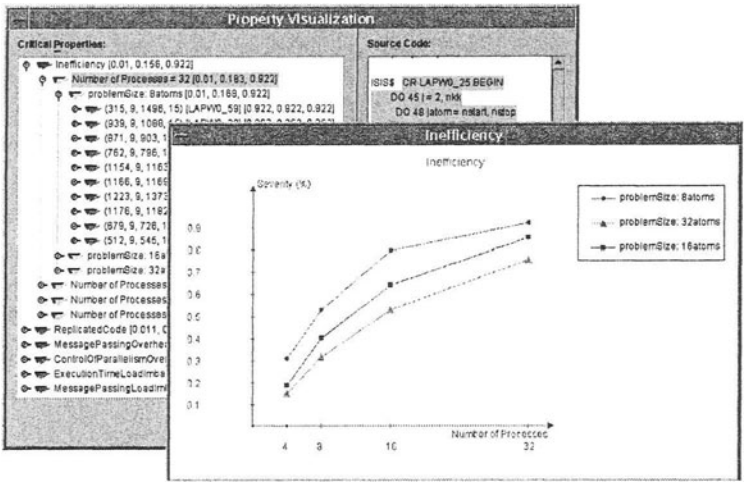


Figure 8. Inefficiency in LAPW0

$\forall j, 0 \leq j \leq z, a_j = a + j * c$. The previous example could also have been written as ("-mp = {1, 2:8:2}")).

7. Experimental Results

In this section, we present a performance analysis for a material science application conducted on an SMP cluster with 16 quad nodes connected through Fast Ethernet. LAPW0 (Linearized Augmented Plane Wave, version 0) [3]calculates the effective potential of the Kohn-Sham eigen-value problem. Implemented as a Fortran 90 MPI code, it has been examined with four problem sizes (representing 8, 16, 32 and 64 atoms) and five machine configurations (1, 4, 8, 16 and 32 CPUs). Due to a lack of memory, however, the last problem size could not be executed with only one CPU. Each line shown in the charts of this section refers to the execution of LAPW0 for a single problem size and different machine sizes. Aksum's user portal shows that, for all problem sizes, the code inefficiency increases for larger machine sizes (see Figure 8). This problem reflects the most important bottleneck of LAPW0, which Aksum indicates by placing the property Inefficiency at the top of the critical property list, as shown in Figure 9. The highest values for this property, however, appear only in the main program, which suggests that there are multiple code regions that are responsible for the inefficient behavior of LAPW0. In fact, Aksum shows that the causes for the inefficiency cannot be explained by a specific but by a combination of several properties (all pictures refer to the main program):

- ReplicatedCode (Figure 10)

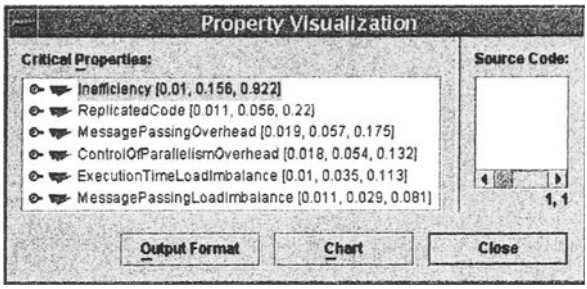


Figure 9. Properties found in LAPW0

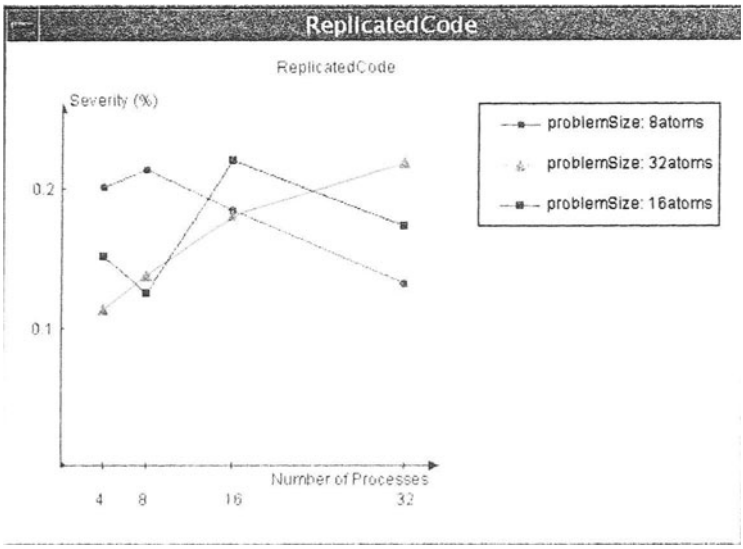


Figure 10. ReplicatedCode in LAPW0

The property *ReplicatedCode* is defined to detect code regions without any overhead (e.g. communication or synchronization) and with approximately identical parallel and sequential execution time. LAPW0 contains several regions that are not parallelized and executed by all CPUs, and the property *ReplicatedCode* was able to detect several of them; the cumulative effect of these regions stands out in the main program, responding for about 22% of the execution time.

- MessagePassingOverhead (Figure 11)

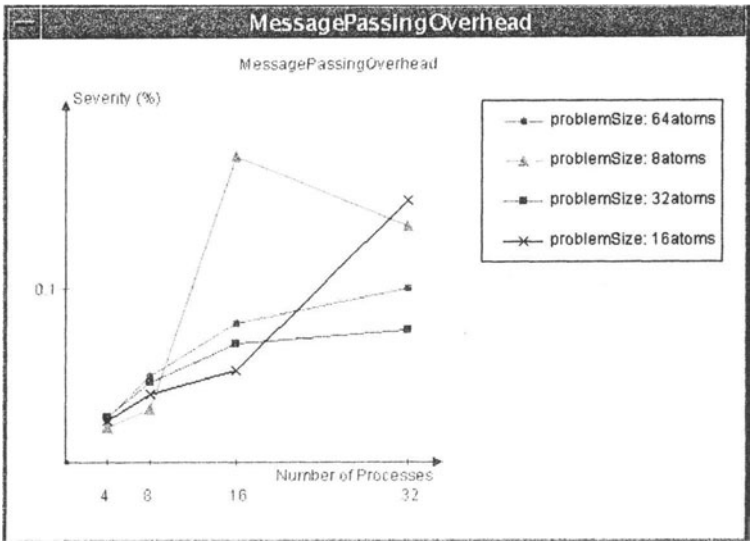


Figure 11. MessagePassingOverhead in LAPW0

SCALEA, the instrumentation system used by Aksum, provides the time a program spends in communication. Therefore, the property *MessagePassingOverhead* only needs to normalize the communication time (using, for instance, the execution time of the main program). The severity of this property increases with larger problem and machine sizes, and since SCALEA uses inclusive metrics, the worst instances of this property always appear in the main program.

- ControlOfParallelismOverhead (Figure 12)

  The overhead caused by control of parallelism (for instance, by routines like MPI_INIT and MPI_FINALIZE) is also directly measured by SCALEA, and like the property *MessagePassingOverhead*, *ControlOfParallelismOverhead* just needs to normalize the value provided by the instrumentation system.

- ExecutionTimeLoadImbalance (Figure 13)

  The code distributes a set of atoms onto the processes, but for the problem sizes tested it is not always possible to distribute them equally onto a set of CPUs. The property *ExecutionTimeLoadImbalance*, which assumes an SIMD application, is able to detect this imperfect work distribution.

These properties, however, are not sufficient to completely explain the inefficient behavior of LAPW0 for some machine and problem sizes, and further

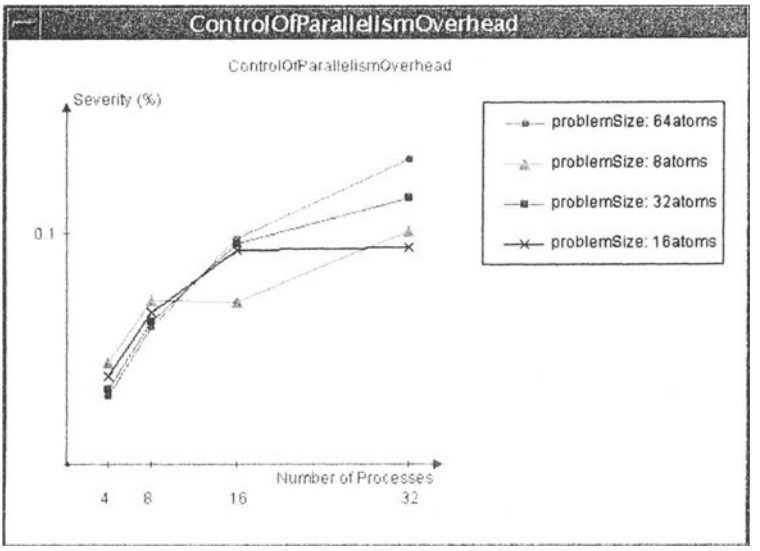


Figure 12. ControlOfParallelismOverhead in LAPW0

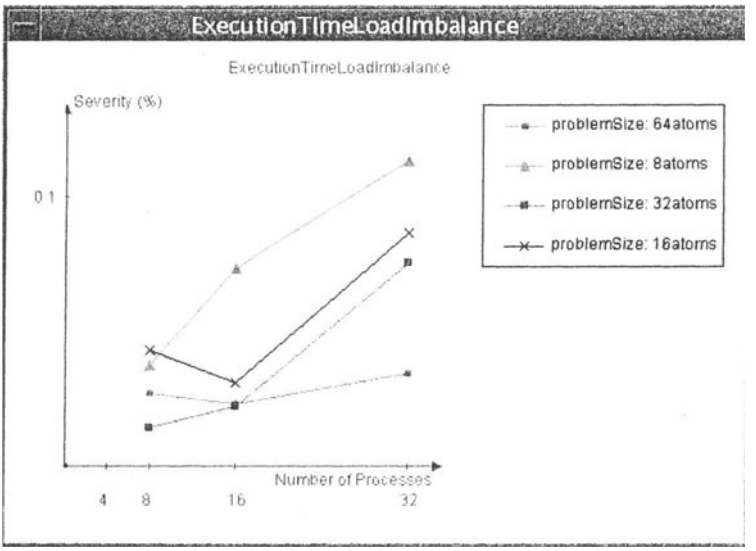


Figure 13. ExecutionTimeLoadImbalance in LAPW0

studies are needed in order to specify and detect other performance problems in this code.

8. Related Work

Various groups have developed performance tools that are oriented towards automatic analysis. Paradyn [10]performs an automatic online analysis by searching for performance bottlenecks based on thresholds and a predefined (but immutable) set of hypotheses.

The European working group APART [6]defined a specification language for performance properties of parallel programs based on which JavaPSL and many Aksum properties have been designed.

Kappa-Pi [4]and Earl/Expert [14]are post-mortem tools that search for performance properties in message passing trace files in combination with source code analysis. Expert also covers OpenMP and mixed parallel programs, and uses the concept of performance properties organized in a hierarchy. Performance properties are also used in the Peridot project [7].

In [13]an approach is described that uses machine learning to detect performance problems in message passing codes. A decision tree which is trained for different target architectures is employed to detect individual communication performance problems.

All of the previously mentioned tools concentrate performance analysis on single experiments only whereas Aksum examines the performance for multi-experiments that analyses a variety of machine and problem sizes. Aksum allows the user to modify and add new properties and property hierarchies. Aksum also provides a rich set of pre-defined properties and several possibilities to impact the search for properties. We believe that Aksum is also more flexible to customize the automatic search for parallel and distributed programs compared to other work.

9. Conclusion

Aksum provides users with automated guidance for discovering performance problems in their applications for distributed and parallel architectures. Aksum is a highly flexible multi-experiment performance analysis tool which is based on a set of problem and machine size values. It tries to automatically detect all critical performance problems. Programmers are guided to performance problems that should be addressed for performance tuning. A very flexible mechanism is provided to customize the properties of interest and to control the search process.

Currently Aksum conducts post-mortem performance analysis. We are working towards a new version that provides automatic performance analysis during

execution of a target application. Moreover, we plan to extend Aksum for Grid applications.

References

[1] OpenMP Website: www.openmp.org.

[2] Siegfried Benkner. VFC: The Vienna Fortran Compiler. In *Journal of Scientific Programming*, 7(1):67-81, 1999.

[3] Peter Blaha, Karlheinz Schwarz, and Joachim Luitz. WIEN97, Full-potential, linearized augmented plane wave package for calculating crystal properties. Institute of Technical Electrochemistry, Vienna University of Technology, Vienna, Austria, 1999.

[4] A. Espinosa, T. Margalef, and E. Luque. Automatic Performance Evaluation of Parallel Programs. In *Proceedings of the 6th Euromicro PDP Workshop*, IEEE Computer Society Press, January, 1998.

[5] T. Fahringer and C. Seragiotto. Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. in *Proceedings SC 2001*, November, 2001.

[6] T. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, and J. Träff. Knowledge Specification for Automatic Performance Analysis. http://www.fz-juelich.de/apart-1/reports/wp2-asl.ps.gz. January, 2001.

[7] M. Gerndt, A. Schmidt, M. Schulz, R. Wismüller. Performance Analysis for Teraflop Computers – A Distributed Approach. *Proceedings of the 10th Euromicro PDP Workshop*, IEEE Computer Society Press, January, 2002.

[8] M. Grand. *Patterns in Java, Volume 1*. Wiley, 1998.

[9] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. In *Parallel Computing*, 22(6):789-828, Sept. 1996.

[10] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, November 1995.

[11] R. Prodan and T. Fahringer. ZENTURIO: An Experiment Management System for Cluster and Grid Computin. In *Proceedings of the 4th International Conference on Cluster Computing (CLUSTER 2002)*, Chicago, USA, September, 2002.

[12] H. Truong and T. Fahringer. SCALEA: A Performance Analysis Tool for Distributed and Parallel Program. In *8th International Europar Conference*, August, 2002.

[13] J. Vetter. Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies. In *Proceedings of the 14th International Conference on Supercomputing*, pp. 245-254, Santa Fe, New Mexico, May, 2000.

[14] F. Wolf and B. Mohr. Automatic Performance Analysis of SMP Cluster Applications. Internal Report, Forschungszentrum Jülich GmbH. August, 2001.

III

GRID PERFORMANCE AND APPLICATIONS

COMMERCIAL APPLICATIONS OF GRID COMPUTING

Catherine Crawford, Daniel Dias, Arun Iyengar, Marcos Novaes, Li Zhang
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
{catcraw,dias,aruni,mnovaes,zhangli}@us.ibm.com

Abstract This paper provides an overview of commercial applications of Grid computing. We discuss Web performance and present a Grid caching architecture. Our Grid caching architecture offloads requests to Grid caches when Web servers become overloaded. We describe performance and traffic modeling techniques which can enhance Grid applications such as caching. We also discuss how Grid computing can be applied to financial applications. A key requirement here is that fast response times are needed. We present a Grid services scheduler that is well suited to commercial applications requiring fast response times.

Keywords: caching, Grid computing, performance modeling, traffic modeling, Web performance

1. Introduction

Grid computing is evolving as the next major distributed computing platform. In the 1990s, large scale cluster computing became a commercial reality [17, 2], with clusters used for both commercial computing, such as for scalable Web Servers [13], and for parallel scientific computing [1]. More recently, cluster computing has evolved to distributed computing on a large scale, across geographic and, in some cases across organizational boundaries, and has been referred to as Grid computing [9]. Many of the early applications of Grid computing were for parallel scientific applications, distributed beyond locally distributed clusters to Grids of multiple clusters or supercomputers across geographically distributed sites. In this paper we examine commercial applications of Grid computing, particularly for Web serving and financial applications.

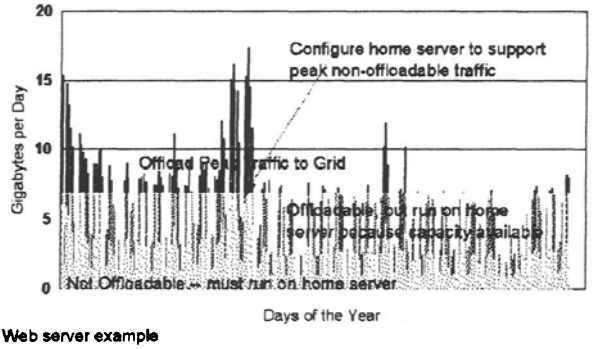
Concurrently with the emergence of Grid computing, the World Wide Web (Web) has been evolving from primarily that of information access in the 1990s, to a Service Oriented Architecture, using the Web Services paradigm. The Web Services protocols [24]that have been defined can use the Web HTTP protocol

for transport, and provide a services-oriented architecture; this includes an interface definition language called Web Services Definition Language (WSDL) [4], for accessing remote services. These two industry trends of Grid computing and Web Services are converging in the recent definition of the Open Grid Services Architecture (OGSA) in the Global Grid Forum [11, 23]. OGSA uses the extensibility defined in WSDL, and specifies a Grid service factory, instance, registry, discovery, life-cycle, service data, among other OGSA service interfaces and behaviors [10].

The initial applications on the Grid were scientific numerically intensive computing applications, that extend the computations from cluster supercomputers to the Grid. For example, the U.K. e-Science Grid research programme (http://www.escience-grid.org.uk/) had an initial focus on bioscience applications (http://www.ercim.org/publication/Ercim_News/enw45/boyd1.html), such as molecular simulation, earth science including climate research and earth observation, and astronomy. The Tera Grid links supercomputing Centers in the U.S, with a focus on open scientific research. One commercially oriented Grid project is the eDiamond Grid for breast cancer screening and diagnosis which is building a "national digital mammography archive for the UK", and a similar mammography Grid with the University of Pennsylvania (http://www.gridtoday.com/02 /1104/100640.html). Another commercial Grid is the Butterfly Grid for multiple concurrent game players (http://www.butterfly.net/). Commercial Grids are in the exploratory stage especially in the financial, life sciences, petroleum, and auto industries.

One of the key motivating factors for using a Grid for commercial applications is illustrated in Figure 1. This figure shows the load on a Web site, in terms of the Gbytes/day served. Also shown is the portion of the data served that is static and can be offloaded to a Web cache; the remaining dynamic data which typically needs to run on the home server is labelled as "non-offloadable". The home server must be configured to handle the peak of the non-offloadable traffic; however, configuring the home server for the peak non-offoadable traffic enables it to serve most or all of the traffic for most of the time, except for peak loads. Thus, one of the key advantages of Grid computing for such commercial applications, is that of being able to configure the home server for the peak non-offloadable traffic, and to be able to offload all of the peak traffic to the Grid. In Section 3 we will describe how this can be achieved for Web serving. Note that typical Web caching services offload all of the cacheable traffic, regardless of whether the home server can handle the traffic or not, rather than offloading only the peak traffic, when needed. In Section 4 we show how financial applications can be speeded up by selectively offloading some of the computation to Grid servers.

One categorization of Grids is in terms of intra-grids, extra-grids and inter-grids. Intra-grids are distributed within an organization. Typically, this would



Web server example

-- For typical Web serving, about 30% was not offloadable
-- Applies to other applications with different fractions offloadable to Grid
-- For technical computing / compute intensive, offloadable fraction often ~ 100%
-- For high update transaction environments, offloadable fraction small

Figure 1. Web traffic, and the proportion which can be offloaded.

be across geographically distributed sites within the organization, though it may be across multiple clusters or servers on a campus. Most of the commercial applications of Grids today are for intra-grids, often across multiple sites of a company, which have peak loads at different times. One principal reason for this is security, especially the data security in an execution environment. In this paper, given the focus on commercial applications of Grids, we will primarily consider intra-grids. Extra-grids open intra-grids to specific trusted partners. For commercial applications, extra-grids could allow the off-loading of peak traffic to a trusted third party, for example to an application hosting service provider. The scientific Grids mentioned earlier fall into this category. Inter-grids are defined as linking multiple Grids, to allow sharing of resources across Grids with dynamic resource discovery. Inter-grids are in the research stage, and there are no examples at this time.

The business value of adopting a Grid model is summarized in Figure 2. First, it can provide for scalability, by allowing the use of under-utilized resources, both within a company in the intra-grid environment, and via outsourcing to an extra-grid. Most compute environments have a peak load much higher than that of the average load, as discussed earlier. The Grid can be used to off-load the peak traffic, either to available intra-Grid resources, or to an extra-Grid. This could lead to significant cost savings by configuring a system for average rather than peak traffic, and through more efficient usage of existing resources. In

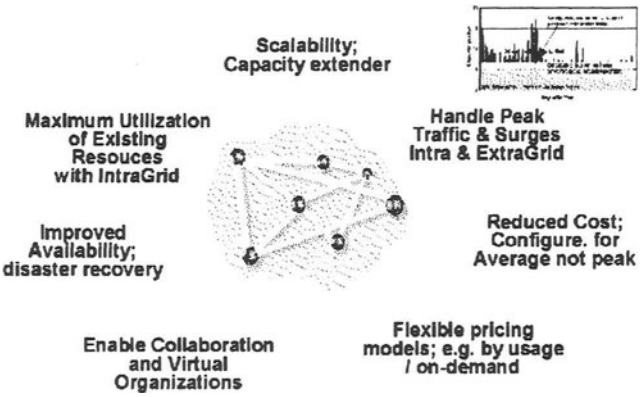


Figure 2. Business value of adopting a Grid model.

configurations with an off-load of peak traffic to an extra-Grid, pricing models based on usage, rather than peak traffic configurations, can also lead to cost savings. Properly configured, a Grid based solution can also provide higher availability and support for disaster recovery.

The remainder of this paper is organized as follows. In Section 2 we describe issues related to high-performance Web serving, a critical commercial application which can benefit from the Grid. In Section 3, we describe how Grid technology can be used for Web caching, especially to handle peak traffic loads. We also discuss how performance modeling and traffic modeling can enhance Grid caching. The use of Grids for financial applications, and a Grid scheduler for these applications, are described in Section 4. Finally, concluding remarks appear in Section 5.

2. High-Performance Web Serving

Web serving is a critically important application that can benefit from parallel processing and Grid computing. Web serving lends itself well to concurrency because requests from different clients can generally be processed independently. A Web site which receives significant traffic would typically contain many servers. The servers might be geographically distributed in order to bring content closer to clients as well as to increase availability.

Requests can consume widely differing amounts of resources to satisfy. If I/O bandwidth is the bottleneck, then large objects become undesirable to serve. Image files can consume significant I/O bandwidth, so limiting the use of images can improve performance considerably. Requests for files are known as static requests and generally consume less overhead than dynamic requests which invoke programs to generate data on-the-fly for satisfying requests. Requests for dynamic data can consume orders of magnitude more CPU time to satisfy than requests for static data. Therefore, even if a Web site serves only a fraction of its requests dynamically, dynamic requests can consume the bulk of the CPU cycles. Static requests are easier to offload to a Grid than dynamic requests.

Encryption can also add significant overhead to a Web site when confidentiality is required. Encryption is typically handled on the Web using the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocol [18, 8]. The SSL protocol requires a handshake at the beginning in order for the client and server to negotiate a session key used for encrypting data via symmetrical cryptography. Session key generation is expensive. The overhead of session key generation is reduced by using the same session key for multiple transactions. In order to limit security exposure, session keys have a limited lifetime after which they must be changed. In Grid computing, security and encryption are often less important for intra-grid environments than for extra-grid or inter-grid environments.

In a scalable Web site, requests are distributed to multiple servers by a load balancer. The Web servers may access one or more databases or other back-end systems for creating content. The Web servers would typically contain replicated content so that a request could be directed to any server in the cluster. One way to share static files across multiple servers is to use a distributed file system such as AFS or DFS [15]. Copies of files may be cached in servers for faster access. This approach works if the number of Web servers is not too large and data doesn't change frequently. For large numbers of servers for which data updates are frequent, distributed file systems can be highly inefficient. Part of the reason for this is the strong consistency model imposed by distributed file systems. Shared file systems require copies of files to be strongly consistent. In order to update a file in one server, all other copies of the file need to be invalidated before the update can take place. These invalidation messages add overhead and latency. At some Web sites, the number of objects updated in temporal proximity to each other can be quite large. During periods of peak updates, the system might fail to perform adequately. Shared file systems would be more appropriate for an intra-grid environment than an extra-grid one.

Another method of distributing content which avoids some of the problems of distributed file systems is to propagate updates to servers without requiring the strict consistency guarantees of distributed file systems. Using this approach, updates are propagated to servers without first invalidating all existing

copies. This means that at the time an update is made, data may be inconsistent between servers for a little while. For many Web sites, these inconsistencies are not a problem, and the performance benefits from relaxing the consistency requirements can be significant. This approach for distributing content is easier to apply to a Grid environment than that of shared file systems.

Load balancers distribute requests among multiple Web servers. One method of load balancing requests to servers is via DNS servers. DNS servers provide clients with the IP address of one of the site's content delivery nodes. When a request is made to a Web site such as *http://www.ibm.com/employment/*, "www.ibm.com" must be translated to an IP address, and DNS servers perform this translation. A name affiliated with a Web site can map to multiple IP addresses, each associated with a different Web server. DNS servers can select one of these servers using a policy such as round robin [3].

One of the problems with load balancing using DNS is that name-to-IP mappings resulting from a DNS lookup may be cached anywhere along the path between a client and a server. This can cause load imbalance because client requests can then bypass the DNS server entirely and go directly to a server [7]. Name-to-IP address mappings have time-to-live attributes (TTL) associated with them which indicate when they are no longer valid. Small TTL values can limit load imbalances due to caching. The problem with this approach is that it can increase response times [19]. Another problem with this approach is that not all entities caching name-to-IP address mappings obey TTL's which are too short. Caching of name-to-IP address mappings is a problem if DNS is used to route request to Grid servers or caches because a Grid server cannot easily be removed; clients may continue to route requests to a former Grid server even if the DNS has stopped sending requests to the server.

Another approach to load balancing is using a connection router in front of several back-end servers. Connection routers hide the IP addresses of the back-end servers. That way, IP addresses of individual servers won't be cached, eliminating the problem experienced with DNS load balancing. Connection routing can be used in combination with DNS routing for handling large numbers of requests. A DNS server can route requests to multiple connection routers. The DNS server provides coarse grained load balancing, while the connection routers provide finer grained load balancing. Connection routers also simplify the management of a Web site because back-end servers can be added and removed transparently.

IBM's Network Dispatcher [12] is one example of a connection router which hides the IP address of back-end servers. Network Dispatcher uses Weighted Round Robin for load balancing requests. Using this algorithm, servers are assigned weights. All servers with the same weight receive a new connection before any server with a lesser weight receives a new connection. Servers with

higher weights get more connections than those with lower weights, and servers with equal weights get an equal distribution of new connections.

With Network Dispatcher, requests from the back-end servers go directly back to the client. This reduces overhead at the connection router. By contrast, some connection routers function as proxies between the client and server in which all responses from servers go through the connection router to clients.

In the next section, we show how load balancing is used to send requests to Web servers as well as to Grid caches which are used when the Web servers become overloaded. The load balancer is configured to route requests to new caches in response to increasing load.

Caching on the Web exists in a number of different forms. Client browsers may cache objects so that they don't have to be fetched from remote sources. Proxy caches are caches which exist on Web proxy servers. When a client connects to the Web, the client typically goes through a proxy server which is shared by many clients. The proxy server may have a cache which is shared by several clients. Static documents which are designated as being cacheable may be stored in the proxy cache when first requested by a client. A subsequent request for the object from another client sharing the cache would obtain the object from the proxy cache instead of going out to the network to fetch the object.

Content distribution networks (CDN) cache content remotely from servers at edges of the network. Web sites will pay to use a CDN to offload requests and move content closer to clients. By contrast, proxy caches function on behalf of clients, and a Web site does not pay to use a proxy cache.

3. Performance Modeling and Web Caching Grids

We propose a *Web caching Grid* in which remote servers on a Grid may function as caches when request rates are high and quality of service (QoS) for response times are in danger of degradation. As illustrated in Figure 1 in Section 1, the fraction of the network bandwidth that can be offloaded from a typical Web server is high. Traffic can be appropriately monitored in order to determine when requests should be offloaded to remote Grid caches. Grid caches would typically store static Web data which does not need to be encrypted. That way, security is preserved, and sophisticated back-end processing is not required for Grid caches to serve data, while response time goals can be maintained.

In this section, we are concerned with the following scenario. A high volume Web site customer can dynamically configure servers (i.e. provision additional servers) with sufficient advanced warning (i.e. minutes to hours) that the demand for capacity is increasing. In order to generate this advanced warning we need an on-line load measurement system, a real time prediction engine, and an event generation mechanism. The addition of forecasted data has the additional

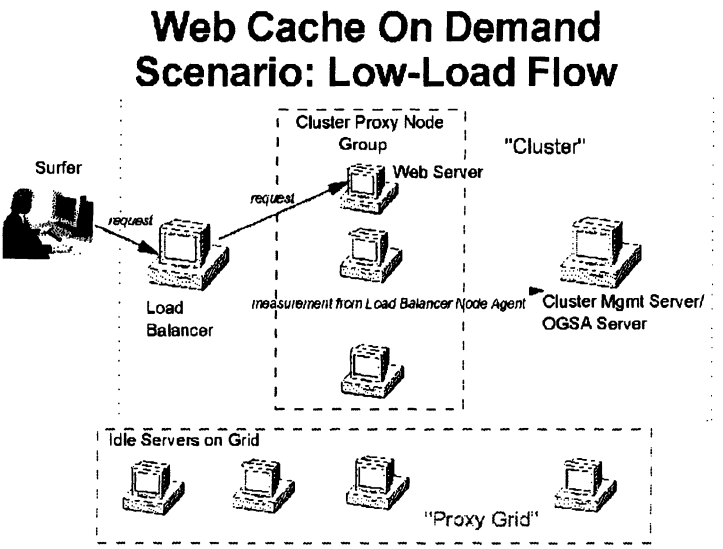


Figure 3. Under low and moderate loads, all requests are sent to Web servers.

benefit of decreasing the frequency with which provisioning, quite possibly an expensive task, occurs since our decision window has increased.

Consider the scenario shown in Figure 3. Here we have a load balancer distributing Web requests to multiple nodes in a Web serving cluster. More precisely, in the N node cluster, m nodes comprise the content server, and $N-m$ nodes act as caching proxies for the server. Furthermore, client machines not included in the cluster, i.e. nodes on the Grid, have registered with the cluster management server to also be proxies for the Web server. We envision that such registration would utilize OGSA-based interfaces and infrastructure.

During normal, or, more precisely, low-load operation, all requests are routed to the configured Web server. Once an overload of requests is predicted, where overload has been defined using known server capacity forecasted arrival rates, and a specified quality of service (QoS), an event is sent to the cluster management server. This predicted overload event is forwarded to a provisioner which can then add a proxy to the cluster (via node and load balancer configuration) to accommodate the forecasted demand without risking the QoS for incoming requests. This scenario is shown in Figure 4.

If all of the proxies in the cluster have been provisioned and increased demand is again predicted (a cluster overload condition), the provisioner now moves to contacting registered proxies in the virtual organization, or Grid, once the overload event is received. This scenario is shown in Figure 5.

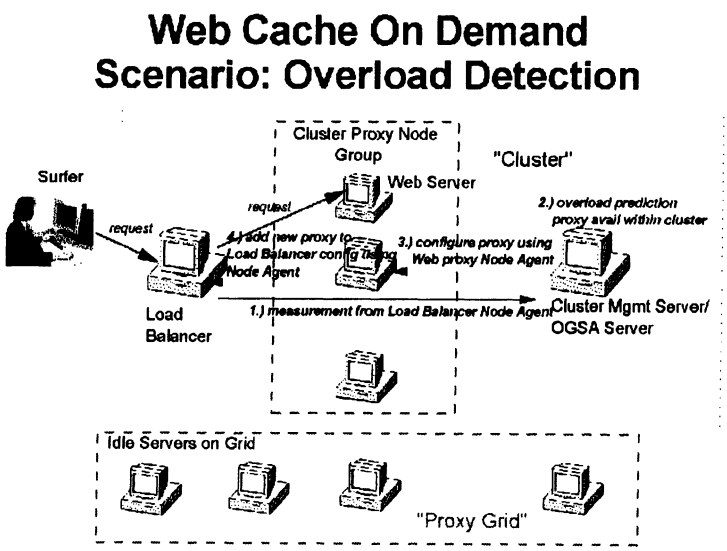


Figure 4. Handling overload when a proxy is available within the cluster.

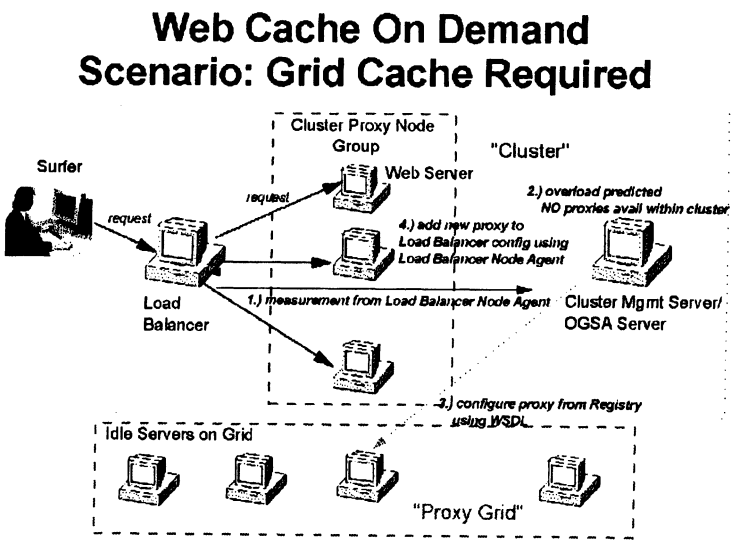


Figure 5. Handling overload when a Grid cache is needed.

As was stated previously, a key component to the caching Grid is on-line modeling and prediction. In the caching Grid that we have built we utilize the eModel tool to fulfill this necessary function. In the following paragraphs we briefly summarize the eModel tool.

In autonomic computing, where each system component monitors its own performance data, and accordingly takes necessary actions, (making it self-healing, self-configuring and self-optimizing) closer integration of various analytic tools for modeling its online performance with the underlying runtime systems is a requirement. In contrast, traditional focus on tooling has been on developing sophisticated off-line or on-line tools capturing as much of the details of an environment as possible [5, 20]. Integrating these types of tools as run-time *components* of a system demands that we pay as much attention to the integration framework as to the models themselves. Ease-of-use continues to remain an important issue, however, not necessarily only as an issue in running the tool by a human operator but also as an issue in setting up the tool for system development/deployment. Here we introduce a flexible architecture and corresponding tool implementation, *eModel*, for such a modeling framework and demonstrate a usage scenario using a Web caching Grid.

An autonomic system component typically monitors and reconfigures itself to comply with service level agreements (SLAs see [16]for an example SLA specification) on its usage, established with the clients of this system. SLAs are established by clients with the service systems in order to receive a guarantee on various service level objectives, e.g., average response time for supported throughput level during certain time periods, availability of services, etc. The eModel architecture is used during all phases of SLA life-cycles: creation, deployment and runtime monitoring and enforcement. In order to establish an SLA, a client needs an understanding of its expected workload, perhaps predicted from past workload history. The data may be available in predefined formats (i.e., as a file or database table). However, when such data are not available *a priori* or an SLA needs to be renegotiated to reflect changing needs, the data may be collected from a running system. As the data collection occurs, workload models are built, and new SLAs can be constructed. We refer to this specific usage of eModel as an SLA Advisor. From a service provider perspective, during deployment and/or for making a commitment to a client SLA, it needs to understand its available capacity, and analyze its risk in accepting this SLA. The eModel tool framework can also be used as a Risk Analyzer. During actual service invocation, the eModel framework can be used to measure and monitor runtime performance, and predict potential violations of service level objectives. We refer to this usage of eModel framework as an SLA Monitor. Note that sophisticated SLA monitoring may involve both computation of aggregated run-time parameters via metric composition (e.g., computing average from individual response times) and/or online prediction of future values of

composed or component parameters. The eModel framework can also be used to further monitor an individual or a collection of resources, to watch for (current or predicted) problem states, e.g., high utilization, system bottlenecks, etc. We will refer to this use of eModel framework as a Resource Monitor. Finally, observed service performance data and/or observed customer workload can be used to adjust risk analysis, and/or to trigger renegotiation of existing SLAs.

In all of these scenarios, the overall analysis/modeling tool needs to support the following features. First and foremost, it needs to provide ease-of-use in setting up the modeling framework, where a model may be a single plug-in or composed from a collection of plug-ins in tandem. From the point of view of on-line integration of this modeling framework with varied data sources (e.g., database tables, online calls to runtime systems, etc.), we need flexible ways of specifying data sources to be used and their access details. Similarly, the eModel framework needs to support flexible ways of delivering analysis data to other runtime system components (e.g., via database, direct calls, etc.). A detailed set up describing the usage of specific model(s) can be specified declaratively using an XML document, which can be created via a GUI. Then a runtime program can parse the XML document and create the corresponding on-line modeling structure and appropriate hooks into measurement and management tools.

A complete review of the eModel framework design and architecture can be found in [6]. Here, we briefly summarize the eModel features which include:

1. a Java and XML based architecture for portability;
2. a Java API for user defined measurement collection (on- or off-line), model functions, and application or system management interfaces (e.g. events, logging, etc.);
3. a run-time manager consisting of an object based container and a thread pool (for multiple workload tracking and modeling instances);
4. a set of static classes and corresponding API for access to measured data and predicted values either from run-time cache or the persisted database; and
5. a plotting tool for real-time visualization of data, predictions and triggered events.

In the remainder of this section, we describe a sophisticated modeling methodology that could be employed within the eModel framework for our Web caching Grid.

Under the general Grid architecture, the dynamic and heterogeneous nature of the available resources makes it a challenging task for the provisioner to make the resource allocation related decisions such as determining the number and type of Grid machines to allocate for the requested service. The un-

derlying mathematical problem is fundamentally difficult due to the complex request arrival patterns and diverse service mechanisms. We need to apply sophisticated statistics and modeling techniques for analyzing the request arrival patterns to the system, forecasting how these arrivals will change over time, and constructing system models for the request service processes on different platforms. COMPASS is a set of tools to help address these issues based on advanced statistics, stochastic processes, queueing, control and optimization theories. COMPASS stands for Control and Optimization based on Modeling, Prediction and AnalySiS. Its main functions include workload characterization, system and application modeling, automatic model building and on-line optimal control.

Request arrival information is passed on to the workload characterization module, which makes predictions of the Web request process based on the past access patterns and the constantly changing volume using time series models. Furthermore, the key characteristics that have strong impact on the server's performance are also extracted by the workload characterization module. Studies [21–22] have shown that the correlation characteristics such as short-range and long-range dependence have significant impact on the response time measures. So does the burstiness characteristics such as the variability and heavy-tailness of the request distributions. The response time measures under long-range dependent and heavy-tailed request processes can degrade by orders of magnitude, and have a fundamentally different decay rate compared with traditional Poisson models. These key parameters including the correlation factor, the marginal distributions, the detected user access patterns, the visit page sequences and think times, and the various matching distribution parameters are calculated by the workload characterization module to establish a complete workload profile to be used as the input to the other modeling modules.

A common approach to model the request serving process in many service systems is to build queueing models [14]. As the user behaviors and the Web service functions become more and more complex, so are the structures of the Web systems. To model how the customer requests, or *transactions*, are served by such complex systems, a single server queue is far from adequate. The system and application module constructs flexible queueing network models to capture the Web serving process. Each of the multiple components within the server system can be represented as a queue or a more complex sub-queueing network. For example, one can use a queue to model the network component within a system, and use a single server queue to model a database, etc. Different routing mechanisms such as round robin and probabilistic routing, and different service disciplines such as processor sharing or priority policies can be used for each queueing or server component within the model to mimic the component's service behavior. Users can be categorized into multiple classes based on their access behaviors. For given routing and service parameters of

such a queueing system, the system and application modeling module readily obtains the performance related measures such as throughput, utilization and response times, by simulations and queueing network theories. The workload profile into the system can be the original as well as the forecasted profile from the workload models. Using markup languages, the constructed models can be easily described and customized for the many different platforms within the heterogeneous environment.

One set of crucial parameters of such queueing network models is the set of service time requirements for different job classes at each server in the model. These parameters can be calibrated ahead of time for different platforms. This process may require considerable time and experience and is difficult to automate. Recent research [25] has provided a general methodology to infer these per-class service time parameters at different servers based on the server throughput, utilization and the per-class end-to-end response time measurements. The service time parameters are the solution to an optimization problem with queueing-theoretic formulas in the objective and constraints. This is the main function provided in the automatic model building module.

Based on the appropriate data of sufficient detail and accuracy, we can construct workload, system and performance models automatically and integrate to complete the control loop. To achieve the given QoS objective, the online optimal control module activates a controller to dynamically change the scheduling and resource allocation policies within the servers based on these models. Furthermore, the optimization functions in the modeling modules map the given QoS requirements into the most cost and operation efficient hardware and software configurations. The results of these actions are reflected from the monitored performance measures. These performance measures together with the changing workload will again influence the control decisions. With all these functions in place, the system is empowered with self-managing capabilities.

4. Case Study: Grid Computing in Finance

In this section, we explore a particular aspect of commercial applications which are common in finance-oriented environments. In particular, we will discuss the demands of applications used in investment banking, such as portfolio pricing and portfolio optimization. These applications involve mathematical operations which are also common in the field of scientific and technical computing, such as Monte Carlo optimization and stochastic modeling. However, there is one key characteristic which is special to the field of finance: the sensitiveness to the response time. In this section we argue that the traditional architecture used in scientific and technical computing is not designed to support the response time requirements in the technical field, and suggest an alternative

architecture that is more suitable to applications which are extremely sensitive to the response time.

The main requirement that drives Grid applications in investment financing can be simply stated: minimal response time. This is a key motivation for moving finance applications to the Grid. The gain in a few seconds in response time represents a significant advantage over a competitor. Therefore, the goal of Grid-enabled applications in finance is to reduce the response time. This goal is usually achieved using code parallelization. Fortunately, most of the mathematical operations used in investment banking are inherently parallelizable. Each portfolio is typically an aggregation of individual items which can be used as parameters to independent calculations. This is usually done using a scatter/gather model, in which operations are scattered to several nodes in the Grid and then a master node performs a gathering operation, and combines the partial results into a portfolio value. Using this technique it is usually possible to reduce the response time by a factor linear to the number of nodes. Linear scaling means that a result that took 60 seconds using one node may take only 6 seconds using 10 nodes. This represents a huge advantage in real time trading.

We now turn our attention to the ways that are available for scheduling the kind of parallel operations just discussed. In traditional high performance computing the parallel scheduling architecture is batch oriented. Consider, for example, that the Globus Toolkit offers several job managers for batch schedulers, such as Load Leveler, Condor, LSF and PBS. This is because Globus was initially developed by the scientific and technical community. Most of the technical applications are batch oriented and are not very sensitive to response time. Also, most units of work (batch) in scientific and technical computing are generally long (several hours or more), and so the response time can be significantly longer than in financial computing..

The batch computing model has a major disadvantage when dealing with the class of short lived operations just described. One such performance impact is the cost of launching and executing a new set of parallel processes for every request. The task of launching a parallel process is very costly, involving the cost of spawning new process instances, the cost of the initialization of the application code itself, and may also possibly involve security-related costs, such as user authentication. In order to avoid such costs, it is imperative that the application code be maintained loaded in the memory of each Grid node, running as a persistent process. Another advantage of using persistent processes is the fact that they provide persistent state for the application. This persistent state can be used, for instance, as a way of caching intermediate results in a pipelined operation. Such pipelined operations (one in which the results of a previous computation are used as input for another computational step of the pipeline) are very common in investment banking applications. The batch

programming model is unable to take advantage of persistent runtime state, and is also not able to be used by interactive applications.

In order to overcome the limitations of the batch programming model, it is necessary to deploy persistent applications, or services. One of the major features of the new Open Grid Services Architecture (OGSA) is the support of persistent Grid services. The OGSA architecture defines a specific interface for lifecycle management of such services which is designed for the management and control of persistent services. The lifecycle interface defines the rules for service instance creation and destruction, which are essential for the management of persistent service instances. This architecture enables a programming model which offers a major improvement over the traditional batch processing model: interactive, persistent parallel services.

The remainder of this section proposes an architecture for deploying persistent parallel services in a Grid. This architecture is implemented in a Grid scheduler prototype we have developed called Topology Aware Grid Services scheduler (TAGSS). The TAGSS architecture is derived from concepts introduced by the OGSA standard. The OGSA standard mandates that Grid services implement the mentioned life cycle interface, which specifies a creation method, also called the factory interface. The TAGSS architecture defines a basic service, which is the TAGSS Container Factory. The TAGSS container factory can construct individual TAGSS containers, or collections of containers named Grid Container Arrays. The Grid Container Array is also a persistent Grid Service, which is in turn a factory for Grid Object Arrays. The Grid Object Array is the main construct which exports the scheduling functionality of the TAGSS architecture.

The Grid Object Array is a construct similar to an object in object-oriented languages such as Java. The difference is that a method invocation on a Grid Object Array results in method invocations in each object in the array. In order to pass the arguments for the method invocation on a Grid Object Array, an auxiliary data structure called the Grid Data Set is used. The Grid Data Set is basically a matrix where each row corresponds to an argument list for a method invocation. There are three different method invocation semantics which can be used when invoking a method on a Grid Object Array:

1 MULTICAST: in this mode the Grid Data Set contains one row, and the method invocation is done using the arguments in the single row to all objects in the Grid Object Array. In other words, all objects execute the same method with the same arguments. This mode is used to synchronize state in the objects in the array.

2 ANYCAST: in this mode the Grid Data Set typically contains many more rows than the number of objects in the Grid Object Array. For example, this mode can be used with a Grid Data Set of thousands of rows and

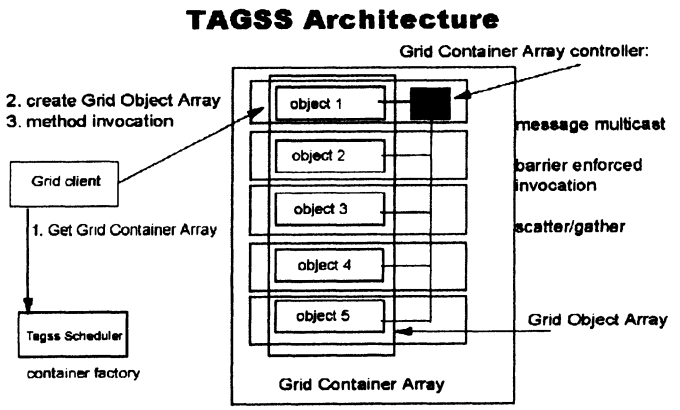


Figure 6. The TAGSS architecture.

a small number of objects in the array. The semantics of ANYCAST dictate that it does not matter which object works on a particular row of the Grid Data Set; any object can work on any row. This method of execution can be used to implement the typical scatter/gather function which is of particular interest in investment banking, as discussed at the end of the section.

3 BARRIER ENFORCED METHOD INVOCATION: this mode of method invocation specifies that there is a one to one correspondence between each row of the Grid Data Set and each object. The objects in the Grid Object Array are indexed, and row zero on the Grid Data Set is used to invoke a method on object zero, row one on object one and so forth. This method is useful when all objects have to complete a specific operation with different arguments, and the invocation is synchronized by a barrier condition.

Figure 6 depicts the basic elements of the TAGSS architecture.

The TAGSS architecture has a unique feature which makes it particularly suitable for Grid deployment. When a Grid Container Array is created, one of the containers is chosen as a coordination point; it creates a small microscheduler object called the Grid Container Array controller. This object is not directly seen or controlled by the client application, and it has the function of enforcing the proper method invocation semantics described above. The microscheduler is especially important in the realization of the ANYCAST method invocation mode, because this mode requires the monitoring of the completion of tasks

(rows in the Grid Data Set) and assigns new tasks (rows) to the objects in the Grid Object Array. This component actually implements a real time scheduling environment.

The TAGSS architecture is well suited for the Grid for a number of reasons. In the first place it creates a distinct Grid Container Array for each client, and therefore each client has its private scheduling domain. This is consistent with the Grid architecture, because the set of resources available to any given user is determined by the privileges associated with the user's Grid certificate. Therefore, the resource set for any given user is potentially distinct, and the scheduling domain is equally distinct. This aspect makes the deployment of a centralized scheduling architecture particularly difficult, because the scheduling procedure would have to consider a different set of resources and constraints for each user. The TAGSS architecture, on the other hand, deploys a microscheduler for every Grid Container Array. It can therefore be considered a fully distributed scheduling architecture, which is more suitable for deployment in the Grid. Another reason that makes the TAGSS architecture appropriate for Grid deployment is that it is designed to be used for the deployment of persistent stateful services, which, as mentioned in the beginning of this section, are necessary to fulfill the minimal response time requirements which are important in commercial Grid applications, and in particular to applications in the field of investment banking.

5. Conclusion

We have presented several issues related to commercial applications of the Grid. Web serving is an example of an application which can benefit from the Grid. We presented a caching Grid which offloads traffic to remote caches when servers become overloaded. Unlike prior caching techniques such as proxy caching and content distribution networks, Grid caches are only used when servers become overloaded.

We described techniques for performance and traffic modeling which can be applied to Grid caches. We also discussed financial applications of Grid computing. A key requirement for financial applications is to have short response times. We presented a scheduler for Grid applications which is specifically targeted to applications which require fast response times.

References

[1] R. Agarwal et al. High Performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2. *IBM Systems Journal*, 34(2):263–272, 1995.

[2] T. Agerwala, J. Martin, J. Mirza, D. Sadler, D. Dias, and M. Snir. Sp2 system architecture. *IBM Systems Journal*, 34(2):152–184, 1995.

[3] T. Brisco. DNS Support for Load Balancing. Technical Report RFC 1974, Rutgers University, April 1995.

[4] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language 1.1, March 2001. http://www.w3.org/TR/wsdl.

[5] Peakstone Corporation. Peakstone eAssurance^{TM} eBusiness capacity management product features, descriptions & benefits. http://www.peakstone.com/pdf/FDB.pdf.

[6] Catherine H. Crawford and Asit Dan. eModel: Addressing the need for a flexible modeling framework in autonomic computing. Research Report RC 22464, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, May 2002.

[7] D. Dias, W. Kish, R. Mukherjee, and R. Tewari. A Scalable and Highly Available Web Server. In *Proceedings of the 1996 IEEE Computer Conference (COMPCON)*, February 1996.

[8] T. Dierksand and C. Allen. The TLS Protocol (RFC 2246). http://www.ietf.org/rfc/.

[9] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, 1998.

[10] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed systems integration. *IEEE Computer*, 35(6):37–46, June 2002.

[11] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002. http://www.globus.org/research/papers/ogsa.pdf.

[12] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network Dispatcher: A Connection Router for Scalable Internet Services. In *Proceedings of the 7th International World Wide Web Conference*, April 1998.

[13] A. Iyengar, J. Challenger, D. Dias, and P. Dantzig. High-Performance Web Site Design Techniques. *IEEE Internet Computing*, 4(2), March/April 2000.

[14] L. Kleinrock. *Queueing Systems, Volume II, Computer Applications*. John Wiley and Sons, 1976.

[15] T. T. Kwan, R. E. McGrath, and D. A. Reed. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer*, 28(11):68–74, November 1995.

[16] H. Ludwig, A. Keller, A. Dan, and R. King. A service level agreement language for dynamic electronic services. In *WECWIS*, June 26-28 2002.

[17] G. Pfister. *In Search of Clusters*. Prentice Hall, 1998.

[18] E. Resorla. HTTP Over TLS (RFC 2818). http://www.ietf.org/rfc/.

[19] A. Shaikh, R. Tewari, and M. Agrawal. On the Effectiveness of DNS-based Server Selection. In *Proceedings of IEEE INFOCOM 2001*, 2001.

[20] Sodalia. NetTraffic^{TM}. http://www.sodalia.com/pdf/nettraffic.pdf.

[21] M.S. Squillante, D.D. Yao, and L. Zhang. Web traffic modeling and web server performance analysis. In *IEEE Conference on Decision and Control (CDC)*, 1999.

[22] M.S. Squillante, D.D. Yao, and L. Zhang. *System Performance Evaluation: Methodologies and Applications*, chapter Internet Traffic: Periodicity, Tail Behavior and Performance Implications. CRC Press, 2000. E. Gelenbe, book editor.

[23] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid service specification. In *Open Grid Service Infrastructure WG, Global Grid Forum*, July 2002.

[24] W3C. Web services. http://www.w3.org/2002/ws/.

[25] Li Zhang, C. H. Xia, Mark S. Squillante, and W. N. Mills III. Workload service requirements analysis: A queueing network optimization approach. In *Tenth IEEE International*

Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2002.

MESH GENERATION AND OPTIMISTIC COMPUTATION ON THE GRID*

Nikos Chrisochoides
Department of Computer Science
College of William and Mary, Williamsburg, VA, USA
nikos@cs.wm.edu

Craig Lee
The Aerospace Corporation
El Segundo, CA, USA
lee@aero.org

Bruce Lowekamp
Department of Computer Science
College of William and Mary, Williamsburg, VA, USA
lowekamp@cs.wm.edu

Abstract This paper describes the concept of optimistic grid computing. This allows applications to synchronize more loosely and better tolerate the dynamic and heterogeneous bandwidths and latencies that are seen in grid environments. Based on the observed performance of a world-wide grid testbed, we estimate target operating regions for grid applications. Mesh generation is the primary test application where boundary mesh cavities can be optimistically expanded in parallel. To manage the level of optimistic execution and stay within the application's operating region, we are integrating grid performance monitoring and prediction into the supporting runtime system. The ultimate goal of this project is to generalize the experience and knowledge of optimistic grid computing gained through mesh generation into a tool that can be applied to other tightly coupled computations in other application domains.

Keywords: mesh generation, Grid computing, optimistic computing, performance analysis

*This work is supported in part by the National Science Foundation under the grant EIA-0203974.

1. Introduction

Computational grids offer a vast pool of computational and storage resources which can be utilized by large-scale engineering and scientific computing applications. The *efficient* utilization of these vast resources, however, can be difficult. Grids provide an environment where resources *and* performance can be dynamic and heterogeneous. It is clear that loosely coupled applications will be naturally "grid-friendly". However, grids offer a way of aggregating more compute power than can be economically done any other way. Hence, there is strong motivation to find ways that enable more tightly coupled applications to effectively utilize grid resources.

To do this, a tightly coupled application must be able to "loosen-up" just enough to tolerate prevailing conditions. This "loosening-up" can be accomplished by relaxing strict synchronization and communication requirements, wherever possible, and by tolerating low bandwidths and high latencies, however possible. Ideally an application should also be able to tell how much "loosening-up" is enough by acquiring the appropriate performance information about the environment and itself. In general, an application's computation-communication ratio should match what the environment can support as much as possible. Many established techniques exist that could be applied here. Such techniques include: partitioning to control the surface area to volume ratio, using clumps (SMP clusters) to increase a host's "cycle density", overlapping communication and computation, pipelining data communication, using shadow arrays or ghost zones, using aggregate communication, using data compression, and tuning the communication protocol.

These issues can also be addressed by *restructuring* or *reconceiving* an application, perhaps even using a different *execution model.* As an example, hierarchical n-body methods achieve much faster speeds than traditional all-to-all methods by allowing accuracy to be traded for performance within a certain bound. Different execution models could include coarse-grain dataflow models, such as workflow or stream programming models, and also *optimistic* or *speculative execution* models.

This paper reports on a new project to investigate all of these issues, and specifically *optimistic grid computation. Mesh generation* will be used as the test application. Mesh generation is an integral part of many important scientific computing applications and is a memory-intensive application with major impact on the overall performance of the end-to-end field simulations that could benefit greatly from a grid platform. Using traditional approaches for generating, partitioning, and placing very large meshes on a grid have two weaknesses: (i) I/O and data movement due to mesh re-partitioning, for adaptive applications, is prohibitively expensive, and (ii) there is a trade-off between the quality of the resulting elements and partitions and the performance of field solvers.

Hence, we have developed methods whereby mesh generation can be "loosened-up" by using *speculative* or *optimistic* execution. This presents three challenges:

1) *Maintain high-quality of the solution*, i.e., elements and partitions in the case of parallel meshing, in order to guarantee high-performance for the end-to-end parallel simulation,

2) *Control the optimistic execution* such that *optimal optimism* is achieved, i.e., prevent excessive optimism whereby too much work must be discarded and any performance benefits are lost, and

3) *Provide a runtime infrastructure and appropriate performance metrics* whereby the proper control decisions can be made.

In this project, we address these challenges by performing the following three tasks using a mesh generation application from fracture mechanics:

- Develop a runtime infrastructure that enables an application to interact with its performance environment while facilitating quick prototyping, experimentation, and evaluation.

- Explore how optimistic execution and other techniques must be controlled to scale-up and enable an application to work in its "operating region".

- Develop and evaluate new network metrics and performance models that cover a wide spectrum of communication characteristics using mesh generation methods to implement properly grid-aware applications.

In the next section, we will introduce the concept of optimistic computation and the behavior that an application must exhibit to benefit from it. This application behavior is tied to the concept of an "operating region" for grid applications based on "keeping the pipes full" and matching the computation/communication ratio to the environment. We then introduce parallel mesh generation methods which can be reconceived as optimistic computations. These methods can be supported by a specialized run-time system described in Section 4 that incorporates grid performance monitoring. We also describe the testbeds that will be used to evaluate these mesh generation methods and optimistic grid computation, in general. We conclude in Section 5.

2. Optimistic Computation for Grid Environments

2.1 General Optimistic Computation

Optimistic parallel computation is very similar to optimistic parallel simulation [19]. In that work, multiple end-hosts simulate in parallel and exchange time-stamped events. If a host receives an event with a time-stamp earlier than

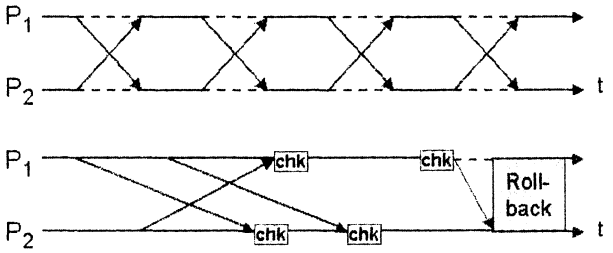


Figure 1. Cost trade-offs between strict control and optimistic control.

its current simulated time, i.e., "in its past", then it must *roll-back* its simulation to the earlier time and start again. This requires that an end-host do *incremental state saving* to enable roll-backs, and possibly send *anti-messages* that undo the effect of messages sent to other end-hosts during the rolled-back period.

The critical trade-off that determines the advantage of optimistic simulation is the overhead of roll-backs (frequency and severity) versus the benefit of more loosely coupled, parallel simulations that have reduced synchronization and communication delays. This is a fundamental trade-off that applies to all optimistic computation. Optimistic execution can reduce synchronization and communication requirements but allows *inconsistent* results to be computed. What is considered inconsistent, of course, is application-dependent. Whatever may be considered inconsistent, an application must detect and correct inconsistencies such that (1) a correct final result is produced, or (2) whatever error may exist in the solution is limited to within some acceptable bound. Again, it is application-dependent whether strict correctness is required or whether bounded error is acceptable.

This trade-off is illustrated in Figure 1. A strictly synchronized application has some overhead due to this synchronization and communication control. An optimistic computation will have some overhead for consistency checking using *validation messages*. Since consistency validation can proceed independently, these messages can be sent asynchronously and also be pipelined. When an inconsistency is detected, however, roll-back must occur which could be a more heavyweight and synchronous operation. For optimistic computation to be advantageous, the following inequality must hold:

$$\text{Cost}_{\text{validation}} + \text{Cost}_{\text{roll-back}} \ll \text{Cost}_{\text{strict-control}} \quad (1)$$

The costs of validation and roll-back should be much less than the cost of strict control. If the costs are similar, or greater, then there will be no advantage, or even a disadvantage. Cost will typically be defined in terms of execution time but this cost could be broken down into communication time, synchronization delays, and also discarded work.

A key issue for this work is "how much optimism is enough"? Optimistic application must have some type of independent parameter, or "control-knob", that determines the amount of optimism. Clearly over-optimistic execution will suffer excessive roll-backs while under-optimistic execution will reduce parallel performance. Besides these application-specific considerations, an application should only generate *just enough* optimism to match the prevailing grid conditions. To understand this issue, we define the notion of an *operating region* for grid applications.

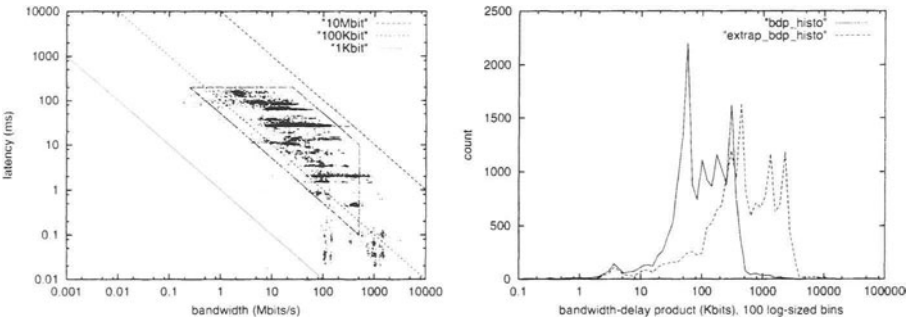


Figure 2. Left: Extrapolated Round-Trip Time latency and Bandwidth with operation region. Right: Bandwidth-Delay Product histograms, both measured and extrapolated.

2.2 The Operating Region for Grid Applications

What properties must an application exhibit to effectively utilize grid resources under the prevailing grid performance among these resources? We can characterize this question on the macro-scale by examining actual performance data from a world-wide grid. As reported in [24], a snapshot of a global grid's performance was used to derive the probable *operating regions* for grid applications based on the number of concurrent threads of computation, size of a "work unit", and the prevailing bandwidths and latencies. This data snapshot captured 17629 unique bandwidth and round-trip latency measurements between 3158 unique host-pairs over 138 unique hosts on four continents. Based on the estimated improvement in processor speeds and network bandwidth in the next ten years, these data were extrapolated to show the expected bandwidths and latencies that will be available in 2010. These data are shown as a scatterplot in Figure 2(left) which shows the *density* of available performance. The three diagonal lines on this graph show the position of three different bandwidth-delay products: 1 Kbit, 100 Kbit, and 10 Mbit. This illustrates the number of bits that could be "in-flight" or "in the pipe" between any source and destination hosts. Figure 2(right) shows the extrapolated increase in the distribution of bandwidth-delay products.

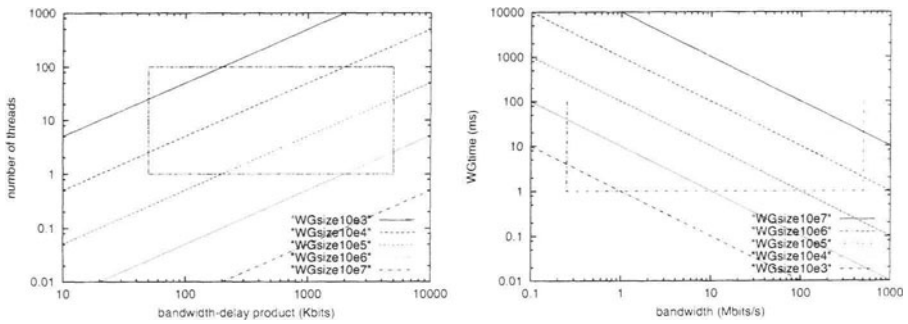


Figure 3. Left: Feasible operating region based on a reasonable number of threads of control and expected bandwidth-delay products. Right: Work Grain time and expected bandwidth.

These results show that since processor speed and bandwidth in a global grid will be increasing, while propagation delays remain constant, that the bandwidth-delay products will be increasing dramatically, i.e., communication "pipes" will be getting "fatter" but not commensurately "shorter". Based on the expected performance density in Figure 2(left), we can draw bounds on where applications should typically operate. Based on the bandwidth, latency and bandwidth-delay products that form this trapezoidal region, we can estimate the *operating region* for grid applications, based on the goals of matching the bandwidth-delay product ("keeping the pipe full") and matching the computation/communication ratios.

These general operating regions are illustrated in Figure 3. Figure 3(left) illustrates the operating region for keeping the pipes full where an application must generate enough independent threads of computation (here 1 to 100) of a given work grain size (in bits) to fall within a given range of bandwidth-delay products (defined by the observed range of bandwidth-delay products in Figure 2). Figure 3(right) illustrates the operating region for matching the computation/communication ratio where the computation time represented by a work grain (in milliseconds) must match the grain's communication time based on its size (in bits) and the available bandwidth (defined by the observed range of bandwidths in Figure 2). (This is only bounded from below since larger work grains will more easily fill the pipe.)

These operating regions are based on a simple, pipeline model of execution and, hence, are only rough estimates since they elide many application issues. Explicit synchronization and data dependency issues are ignored which can clearly affect an application's ability to "hit" the operating region. Most applications will also have a distribution of work grain sizes and computation times, rather than a single, fixed value. Nonetheless, these operating regions capture

the fundamental behavior that applications must exhibit and give us a target for evaluating application performance under optimistic execution.

3. A Case Study: Optimistic Mesh Generation

Whether grid applications can in general be effectively scaled-up to work in these operating regions is an open issue. In essence, the *range of bandwidths and latencies* that an application must tolerate will be greater than in any other environment. Clearly, to be grid tolerant, applications will have to be designed such that they are, in effect, less tightly coupled. In the case of mesh generation, the Parallel Bowyer-Watson method offers an ideal test case. The *Optimistic Delaunay (OD)* approach effectively decouples adjacent mesh regions by allowing a processor to *asynchronously* proceed with remote cavity expansion. The drawback is that some optimistic cavity expansions will be incompatible and will have to be *rolled-back*. Of course, the goal is to minimize the roll-backs and maximize the net performance gain for the application. While we are studying optimistic grid computation in the context of mesh generation, the ultimate goal is to quantitatively understand these techniques and generalize them to other application domains.

3.1 Parallel Mesh Generation

Parallel mesh generation methods decompose the original meshing problem into smaller subproblems that can be solved (i.e., meshed) in parallel. The requirements for the solution of the subproblems on grid environments are: (1) *stability*, distributed meshes should retain the good quality of elements and partition properties of the sequentially generated and partitioned meshes, (2) *tolerance of long, variable, and unpredictable latencies*, (3) *scalability* (4) *fault-tolerant*, and (5) *code re-use*, in order to leverage the ever evolving and maturing basic scalar meshing techniques.

This project is focused on the design and implementation of a *stable, scalable, and latency tolerant* mesh generation methods for grid environments. Fault-tolerance and code re-use are equally important issues and will be addressed in subsequent projects. Mesh *stability* is an important requirement for the accuracy and cost effectiveness of grid-aware Partial Differential Equation (PDE) simulations. *Latency tolerance* is important for the *scalability* of parallel applications, because communication and synchronization latency is the performance bottleneck on grid environments.

3.2 Mesh Generation Methods

We are focusing on Delaunay triangulation methods [2, 17, 29]because they can guarantee mathematically the quality (and thus the stability) of the triangles [31, 8]and tetrahedra [8, 34]. By definition a triangulation T is called Delaunay

if for each element $e \in T$ the open circumscribed sphere of e is empty i.e., none of the mesh points of the mesh are contained within the circumcircle (or circumsphere) of triangles (or tetrahedra) of the mesh. This criterion is called the *empty sphere* or *Delaunay* criterion.

The Delaunay criterion has been used successfully, for sequential mesh generation of complex geometries, since the late 80's. There are many different Delaunay triangulation methods [17], but the most popular Delaunay meshing techniques are the incremental methods. Incremental methods start with an initial mesh (usually a boundary conforming mesh, see mesh M_o, Figure 4) which is refined incrementally by inserting new points. Each new point is reconnected with existing points of the mesh in order to form a new mesh. The difference between the various Delaunay incremental algorithms is due to: (1) different spatial point distribution methods for creating the new points and (2) different local reconnection techniques for creating the triangles or tetrahedra.

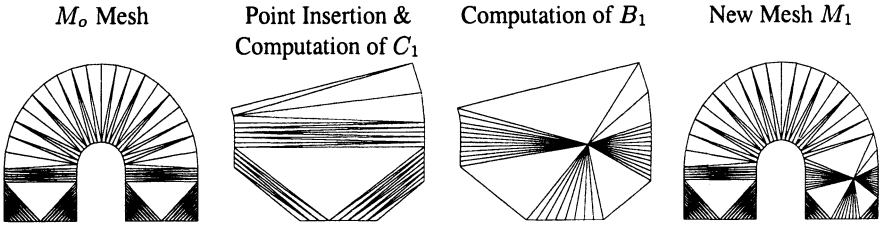


Figure 4. Point insertion and element creation steps of the BW algorithm; initial and final meshes at each iteration shown at the leftmost and the rightmost columns respectively.

The two most popular local re-connection methods are the flip edge/face methods [23]which are difficult and expensive to parallelize [9]and the Bowyer-Watson (BW) kernel [4, 37]. The BW kernel is an iterative procedure: at each iteration, an existing Delaunay mesh, M_i is refined and thus a new M_{i+1} mesh is generated by inserting a new point p_i into M_i after recovering the Delaunay property of the mesh M_i through the local transformation: $M_{i+1} = M_i - C_i + B_i$. Figure 4 depicts the $1st$ iteration of the BW kernel.

The implementation of the guaranteed quality BW kernel, for 3-dimensional geometries, is not an easy task [34], there are very few codes and none in the public domain to the best of our knowledge. The task of implementing a stable and latency tolerant parallel version of the BW (PBW) kernel is one of the key challenges of this project.

In the PBW kernel, each processor concurrently refines the mesh by inserting new points and re-triangulating their cavities. A pair of concurrently expanding cavities are related to each other in two possible ways:

$$\underline{Case\ I}: \quad C_p \cap C_q = \emptyset, \ p \neq q \tag{2}$$

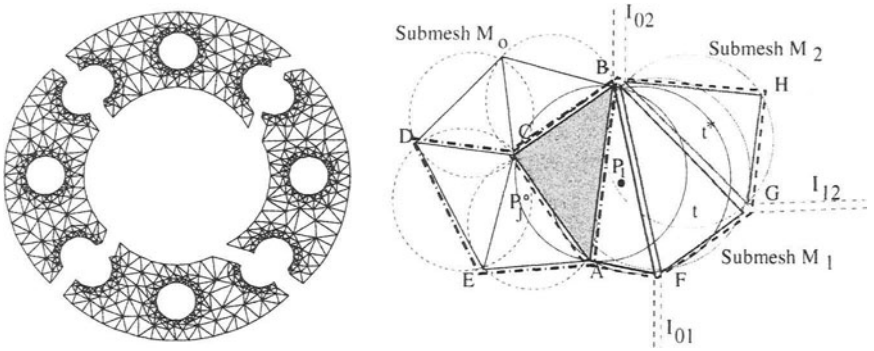


Figure 5. Left: Submeshes of a triangulation and their inter-submesh interfaces. Right: Cavity expansion over more than one submesh. The tetrahedra $t \in M_1$ and $t^* \in M_2$ are non-Delaunay w.r.t point $P \in M_0$; thus the cavity C_P is an interface cavity. Dashed lines show the inter-submesh interfaces.

The cavities do not intersect and thus they can be re-triangulated concurrently. Without lost of generality consider an initial Delaunay mesh, M_o, and two new points p, q within the convex hull of M_o such that $C_p \cap C_q = \emptyset$. Then by the BW kernel and the fundamental Delaunay Lemma [17]we have that the new meshes $M_{p,q}$ and $M_{q,p}$ are Delaunay triangulations and they are the same: $M_{p,q} = M_p - C_q + B_q = (M_o - C_p + B_p) - C_q + B_q = (M_o - C_q + B_q) - C_p + B_p = M_q - C_p + B_p = M_{q,p}$ —as long as the points in $M_o \cup \{p, q\}$ are in general position— and thus the points p and q can be inserted concurrently.

$$\underline{Case\ II}: \quad C_p \cap C_q \neq \emptyset,\ p \neq q \tag{3}$$

The cavities intersect and have to be re-triangulated in a way that non-conformity (see Figure 5(Right)) and/or instability issues like non-Delaunay mesh generation are prevented. In the case $C_p \subset S_i$ for some p, and i, the cavities are computed and triangulated automicaly. Therefore the case $C_p \cap C_q \neq \emptyset$, $p \neq q$ it takes place only if a cavity C_p of a submesh say S_i intersects with another submesh S_j, in this case the cavity expansion interrupts and the cavity remains active (i.e., locks all of its elements) until non-local elements (using remote gather operations) are collected. We refer to these cavities as active interface cavities. While an active interface cavity is expanding remotely the control is released to another local cavity expansion, C_q, (i.e., thread) and thus the cavity C_q might intersect with the cavity C_p.

There are four different approaches to solve the problem of generating unstable (i.e., non-conforming or non-Delaunay) meshes in the context of concurrency: (i) allow remote cavity expansion but impose a partial order between intersecting cavities using a parallel Optimistic Delaunay (OD) meshing, (ii)

mathematically guarantee the uncoupling of submeshes so that non-interface and non-intersecting cavities are computed first and compute interface cavities at the end; this leads to the parallel Delaunay Uncoupling (DU) meshing method, (iii) instead of expanding interface cavities use Constrained Delaunay Triangulation (CDT) techniques, and (iv) ignore interface cavities and compute only the local portions of the cavities, this leads to a non-Delaunay mesh; then using Domain Decomposition (DD) one can reduced the parallel meshing problem into many unrelated sequential problems. Clearly, these four methods vary in terms of stability and communication characteristics which make them an interesting test-case for grid environments.

The main focus of this project is the OD method. The OD method proceeds with the remote cavity expansion, but in order to avoid generating a non-conforming mesh and/or a conforming but non-Delaunay mesh, it imposes a partial order on the triangulation of interface cavities. The remote cavity expansion of interface cavities is a source of very large communication overhead as well as variable and unpredictable communication latencies due to remote data gather operations. Figure 6 shows the time and message size distributions for generating meshes for a "Tee" geometry on a cluster computer with 16 processors. The per-node time distribution for meshing operations is given for generating a mesh of two million elements. The message size distribution is given for generating meshes of three different sizes, including two million elements. The PBW algorithm in its effort to tolerate the long and variable communication latencies of remote interface cavity expansions, uses an *optimistic* or *speculative execution model* i.e., it inserts new points on demand and expands their cavities — otherwise it would be waiting for the completion of remote part of interface cavity expansions. However, for stability reasons, some of the cavities terminate before they re-triangulate and free all of their elements. We call this form of a roll-back in the cavity computation a **setback** in the progress of the PBW kernel.

Such setbacks or roll-backs are a fundamental issue in all optimistic or speculative computational techniques. That is to say, optimistic execution can provide a significant gain in performance only if setbacks do not become excessive. When faced with poor communication (as in a grid), optimistic execution allows a computation to proceed in a less synchronous manner, thus enabling higher parallelism and utilization. If excessive optimism produces excessive setbacks, however, then any gains in performance may be lost.

Hence, a major goal of this project is to quantitatively study the use and control of *optimal optimism* in grid computations, i.e., optimistic execution that does not excessively waste cycles in communication and synchronization delays but also does not excessively waste cycles in setbacks that represent discarded work. We are doing this by modifying the PBW kernel such that we can control the degree of optimism and setbacks — which depend on the

number of outstanding interface cavities, and on newly inserted points in the presence of interface cavities. Determining *where* this optimal optimism occurs, however, is a major challenge. More importantly this control of optimism in PBW will be driven, in part, by the prevailing network conditions. That is to say, just enough optimism will be used to increase parallelism and utilization and overcome insufficient communication bandwidth and latency. For grid applications, such as mesh generation, the *operating region* concept defined in the previous section can be used to determine how optimistically the application can and should be. In essence, a grid application must provide just enough parallelism to keep the network pipes full *and no more*. Beyond this point, an application must be able to execute more asynchronously, or optimistically. Hence, we will use experimental network performance monitoring techniques to gain the information necessary to control the level of optimism in PBW and the demand for communication such that its overall performance remains in its operating region.

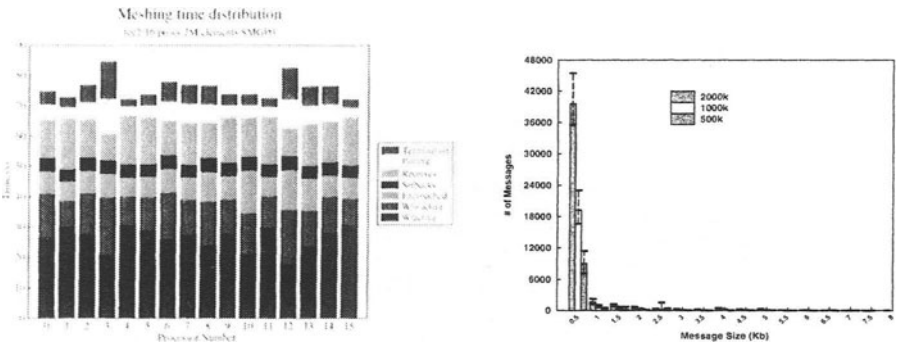


Figure 6. Left: Time breakdown, in percentages, incurred by the PBW algorithm using the OD method, for generating two million element meshes of a 3-dimensional Tee geometry on a 16 computers connected through Fast-Ethernet. Right: Distribution of message sizes for generating 3-dimensional Tee geometry meshes of three different sizes.

4. Supporting Optimistic Computation on the Grid

Enabling an optimistic application such as mesh generation to execute effectively in a grid environment will require some *control mechanism* that can (1) monitor the current grid performance, and (2) interact with the application to help it reach the operating region. To this end, we now describe a *runtime system* that will contain a control module for optimistic execution that utilizes current performance monitoring information. We also describe the testbeds that will be used to evaluate the potential benefits of optimistic grid computation.

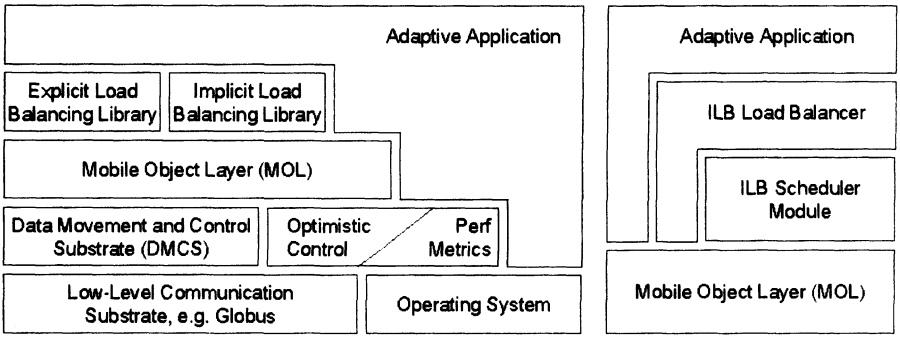


Figure 7. The Runtime System Architecture.

4.1 The Runtime System

The architecture of the runtime system we are using is shown in Figure 7. This leverages existing work on the Portable Runtime Environment for Adaptive Applications (PREMA) for the Crack Propagation Project [6]and Remos [11, 26, 25]. We have ported the Data Movement and Control Substrate (DMCS) [13]on top of Grid communication substrates like those provided as part of Globus [35, 15]. The Application Program Interface of the Mobile Object Layer (MOL) is also being extended in order to permit meaningful applications queries on the characteristics and condition of the underlying Grid environment. Such queries will be performed by higher runtime system support libraries like Implicit Load Balancing (ILB) and Explicit Load Balancing libraries.

This will allow us to explore two approaches regarding the utilization of grid performance information: (1) an explicit approach providing an API to the application to directly use performance information, and (2) an implicit approach where a control module uses the information to control aspects of the application, such as the automatic scheduling of applications tasks. Specifically an *Optimism Control module* will monitor the current application behavior (e.g., number and size of messages, task compute times) and current grid performance (e.g., achievable bandwidths and latencies among end-hosts in use). Based on the operating region model described earlier, the control module will then be able to determine the most appropriate level of optimism and inform the application accordingly.

This runtime architecture approach will have several additional important benefits:

- *Ease-of-Use:* For applications written using the programming model provided by the MOL, the move from clusters and MPPs to grid environments should be a straight-forward evolutionary step.

- *Flexibility:* The runtime will provide a load balancing interface used with a wide variety of scheduling algorithms. Figure 7 also depicts the Scheduler module in the overall architecture. By replacing the Scheduler, applications we will be able to customize the overall load balancing policy without needing to modify any application code. This allows application developers to quickly and easily experiment and find the most effective load balancing method for a given grid environment.

- *Low Overhead:* The overhead incurred by the runtime system will be kept as low as possible. While an application may implement any desired scheduling algorithm, the scheduler should not have to conform to an interface that precludes high performance.

4.2 Performance Monitoring

The research described in this section focuses on what information is required, how resource information is now gathered, and the new developments that must be made to fully meet the needs of optimistic grid computation.

4.2.1 Network Measurements. In a general sense, we are interested in the bandwidth and latency of the portions of the network connecting the resources forming the grid. Specifically, we are interested in the end-to-end latency of each possible connection as well as capacity and availability of the paths between endpoints.

The simplest measurement technique is application level, where the performance of the network is measured by the performance an application receives. Typical application-level approaches rely on actively generating a TCP flow and monitoring its performance [36, 38, 27]. The result is time-averaged available bandwidth over the interval of the communication.

Packet train approaches attempt to measure more fundamental characteristics of the network traffic [7, 22, 21, 20]. Rather than send an actual data stream, a series of packets, ranging from two to tens of packets, are sent between endpoints as quickly as possible. Based on their separation at arrival time, properties of the path at the time that train was sent can be determined. Although this method is most easily used to determine capacity, some information about the link availability may also be obtained.

Either approach can be used in either an active or a passive form. Systems have been developed to passively monitor entire application streams [33, 10]. They have also been developed to monitor for individual packet pairs exchanged by actual applications, rather than generating artificial packet pairs [21].

4.2.2 Predicting Application Performance. The key issue is predicting an application's performance. While past performance is of significant

interest to application performance tuners or network engineers, for grid-based distributed applications short-term prediction is the most important feature. Applications with lifetimes of minutes or hours need predictions over their execution, or even just the next few timesteps.

Performance prediction for grid applications is typically done using time-series models [3, 39, 12, 25, 5]. The output of a time series model is a series of predictions for the expected value of the metric for the future. Equally important, however, is the variance and error in the signal. In particular, for fine-grained applications, the short-term variance in the latency or available bandwidth can be more important than the expected value.

What is the relationship between the metric and the application's performance? In a simple case, the metric may measure exactly what the application does. For example, if an application makes a bulk TCP data transfer across the network, then a simple bulk TCP measurement will be completely accurate. However, using bulk TCP measurements to predict the loss rate of streaming video is quite challenging, and rarely reflect performance of short-term applications [14].

What metrics are needed to predict an application's performance? The shorter the message sent by an application, the more fine-grained the measurement information must be to accurately predict the performance that particular message will receive. Knowledge of the short-term variance in message latency will be important to properly tuning a "loosened-up" application for its execution environment. Furthermore, the performance that an application receives from the network is heavily influenced by how that application uses the network. An application that sends a continuous stream of data may receive different service from TCP than an application that sends only occasional bursts of data, as the bursty application may never force the TCP window open enough to fully take advantage of the bandwidth available on the network.

As the issue of finer-grained application and performance monitoring is approached, the question of traffic predictability appears. The self-similar nature of network traffic has caused some to label network traffic inherently unpredictable. The actual usefulness of predicting network traffic is somewhat more complex, as different network environments have extremely different types of cross traffic. For example, a router processing very bursty WAN traffic may have its queue length vary from empty to overflowing over very short timescales. An Ethernet LAN may have a utilization under 5% 99.97% of the time, and be totally saturated the other 0.03% of the time as a machine performs a high-speed transfer. Neither of these environments has classically "predictable" performance, but it is possible to make meaningful statistical predictions about each environment over the timescale of our applications. It may be completely impossible to predict the short-term behavior of the bursty WAN, but over a longer time interval it may always be possible to transfer significant data across

the WAN. Similarly, it may not be possible to determine when the bursts on the LAN will arrive, but an application may make use of the knowledge that only one out of every 3000 messages will experience congestion. A number of researchers have studied the statistical effect of congestion on TCP flows [32, 16–1], but the effect on individual flows remains an interesting research area.

Predicting application performance cannot be done properly unless predictions are available for the network behavior over the time-scale to which the application is sensitive [18]. Those predictions will only be available if metrics are selected that are capable of recording performance on the time-scale needed by the application. Once those metrics are available, the prediction service must be able to take the measurements and report both the expected value, but also the expected variability in the measurements.

For the parallel mesh generation problem, we are first relying on the models of the DO, CDT, DD, and DU implementations. The amount of data available per processor and the probability of successful speculative execution being rolled back dictate the latency that can be tolerated during data fetches (such as illustrated in Figure 6). With this knowledge, the variability of the network, and the variability of the communication pattern and bandwidth, a distribution of the latencies for the remote requests can be calculated. If this expectation exceeds the tolerable latency, then a different algorithm will have to be selected.

4.2.3 Proposed Network Measurement Approaches. Currently available systems and software do not provide the temporal resolution or variability information required for running fine-grained applications in grid environments. There are, however, measurement techniques that have been developed by the networking community to monitor network behavior that may be appropriate for use supporting these new applications.

We are pursuing three approaches for improving the granularity of network measurements and predictions. The first approach is to add additional instrumentation to currently used network monitoring tools, such as iperf. Iperf is already capable of reporting its performance at regular intervals, although at a coarse scale. However, much information is being lost in the TCP stack by only reporting user-level performance. By adding additional OS-level instrumentation, such as recording packet loss and delays introduced in packet pairs, we will collect more information without additional overhead because we will use same probes already in use by performance measurement systems. Collecting this information and making it available at user level through tools such as NWS and Remos will allow applications to benefit from the knowledge.

The second approach will be to integrate established packet dispersion-based measurement approaches (cite nettimer, cprobe, pathchar, pathload) into the toolchains of existing grid measurement systems such as NWS and Remos. Furthermore, we are exploring integrating passive approaches into these sys-

tems. By combining their existing active approaches with passive measurement options, we can maintain a constant stream of measurements for each path whether the application is currently using the path or not, without creating contention for that path and diminishing the performance of our application.

The third component of this approach is the process of making variability and confidence of prediction first-class information at the application level. In our previous experience developing the Remos toolkit, network variability was discussed, but never fully implemented. Similarly, NWS does not address this issue fully. We will propose modifications to existing measurement APIs to offer more detailed representations of metric variability and prediction confidence than is currently available. We will also enhance the ability of applications to request performance information over specific intervals, for example, specifying that it is interested in the variability of network performance at 250ms intervals for the next 20 minutes.

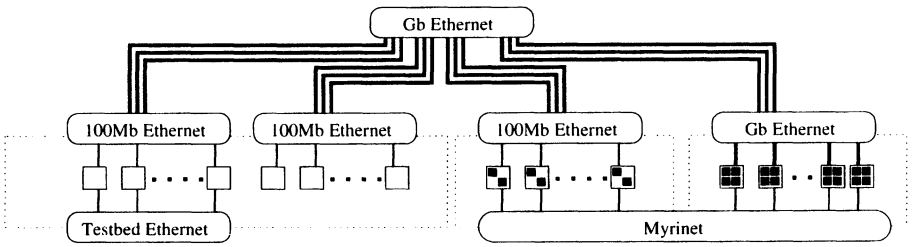


Figure 8. The SciClone cluster. Consists of 64 single processor, 32 dual processor, and 8 quad processor nodes tied together with a fat tree switched Ethernet, as well as Myrinet and a networking testbed. High-performance parallel fileservers are also provided. SciClone's testbed offers opportunities to experiment with the system under controlled conditions, while the cluster is a valuable grid computing resource in its own right for production experiments.

4.3 Evaluation Testbeds

4.3.1 The Cluster Testbed. Initial implementation and experimentation with the optimistic scheduling techniques described here are being performed on the networking testbed portion of the SciClone testbed, shown in Figure 8. The networking testbed provides a 32-node reconfigurable network with production endpoints on which the application can be run under a variety of conditions. This reconfigurability allows us to control the bandwidth and latencies observed by applications by using tools such as NistNet [28]and DummyNet [30].

After initial development on the testbed, the remainder of the SciClone cluster can be used for production runs and reconfigured for brief experiments with limited resources.

4.3.2 The Grid Testbed. The ultimate goal of this project is to demonstrate latency tolerant techniques for applications, such as mesh generation, in a grid environment. After initial development and testing have been done in a more controlled cluster environment, we will proceed with testing in a real-world grid. The exact configuration of the grid resources used is flexible. The general plan, however, is to use grid resources that provide a developmental sequence:

- *Within One Institution:* Using an internal grid for initial testing has the advantage that all resources are co-located and in the same administrative domain. While the number of resources on the grid will be small and close together, our test applications will have to contend with competing traffic for both bandwidth and cycles.
- *Within the Continental U.S.:* A more widely distributed environment will force applications to contend with general Internet traffic.
- *World-Wide:* Institutions from different continents will provide a truly world-wide latency environment in which to demonstrate the potential performance benefits for optimistic computations such as mesh generation.

The goal of this work is to demonstrate superior performance for grid applications using our techniques. To achieve this goal, we will integrate our application tools will all necessary grid services. This will certainly include the use of certificates for security and authorization for using pre-arranged grid resources and code for interacting with Grid Resource Access Managers. It is possible that an information service, such as the Globus MDS, will be used to enable daemons in the DMCS, the Mobile Object Layer, and experimental network measurement tools to discover and interact with one another, but for initial experimental purposes, it may be sufficient to use manual configuration.

5. Summary

We have described the concept of *optimistic grid computation* as a technique to "loosen-up" tightly coupled applications and potentially make them more suitable for execution on grids. While loosely coupled applications are naturally more "grid-friendly", there are clear scientific and economic motivations to make all applications more "grid-tolerant". The goal of optimistic computation is to allow applications to synchronize more loosely and better tolerate the dynamic and heterogeneous bandwidths and latencies that will be seen in a grid environment. Based on the observed performance of a world-wide grid testbed, we have estimated target *operating regions* for grid applications derived from the application's level of concurrency, communication/computation ratio, and the bandwidth-delay product it is currently experiencing.

Mesh generation is our primary test application. We are focusing on Delaunay triangulation methods, and specifically the parallel Bowyer-Watson algorithm. This allows boundary cavities to be expanded optimistically in parallel. When an optimistically generated cavity is found to violate the Delaunay criterion, a setback has occurred and the mesh cavity must be corrected.

To manage the level of optimistic execution and hopefully stay within the application's operating region, we are integrating *grid performance monitoring and prediction* into the Data Movement and Control Substrate. Using packet train measurement techniques, we are also addressing the issue of *network performance variability*. This information will be used by an Optimistic Control module within the Data Movement and Control Substrate.

Finally, the experience and knowledge of optimistic grid computing gained through using mesh generation will be generalized into a tool that can be applied to other tightly coupled computations in other application domains. This will enable a much larger segment of computational science to effectively realize the potential of grid computing. The resulting research findings and runtime infrastructure should be of interest not only to computational scientist using parallel meshing, but to a broad spectrum of computational scientists and engineers who want to use grid environments for tightly coupled applications.

References

[1] Eitan Altman, Kostia Avrachenkov, and Chadi Barakat. A stochastic model of tcp/ip with stationary random losses. In *Proceedings of ACM SIGCOMM 2000*, 2000.

[2] T. Baker. Delaunay triangulation for three dimensional mesh generation. Technical Report 1733, Princeton University, MAE, 1985.

[3] Sabyasachi Basu, Amarnath Mukherjee, and Steve Klivansky. Time series models for internet traffic. Technical Report GIT-CC-95-27, Georgia Tech, 1995.

[4] Adrian Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[5] George E. P. Box, Gwilym M. Jenkins, and Gregory Reinsel. *Time Series Analysis: Forecasting and Control*. Prentice Hall, 3rd edition, 1994.

[6] Bruce Carter, Chuin-Shan Chen, L. Paul Chew, Nikos Chrisochoides, Guang R. Gao, Gerd Heber, Antony R. Ingraffea, Chris Myers Roland Krause and, Démian Nave, Keshav Pingali, Paul Stodghill, Stephen Vavasis, and Paul A. Wawrzynek. Parallel fem simulation of crack propagation – challenges, status. In *Lecture Notes in Computer Science*, volume 1800, pages 443–449. Springer-Verlag, 2000.

[7] Robert L. Carter and Mark E. Crovella. Measuring bottleneck link speed in packet-switched networks. *Performance Evaluation*, 27 and 28, October 1996. also appears as Boston University BU-CS-96-006.

[8] P. Chew. Guaranteed-quality triangular meshes. Technical Report TR 89-983, Cornell University, Department of Computer Science, 1989.

[9] Nikos Chrisochoides and Démian Nave. Parallel delaunay mesh generation kernel. *IJNME*, To appear, 2003.

[10] J. G. Cleary and H. S. Martin. Estimating bandwidth from passive measurement traces. In *Passive and Active Measurement Workshop (PAM-2001)*, 2001.

[11] Peter Dinda, Thomas Gross, Roger Karrer, Bruce Lowekamp, Nancy Miller, Peter Steenkiste, and Dean Sutherland. The architecture of the remos system. In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, August 2001.

[12] Peter A. Dinda. *Resource Signal Prediction and Its Application to Real-time Scheduling Advisors*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 2000. Available as Carnegie Mellon University Computer Science Department Technical Report CMU-CS-00-131.

[13] Andriy Fedorov. Runtime substrate for grid-aware adaptive computations. cs710 Project Report, Computer Science Department, College of William and Mary, Williamsburg, VA 23197, Spring 2003.

[14] Anja Feldmann, Anna C. Gilbert, Polly Huang, and Walter Willinger. Dynamics of IP traffic: A study of the role of variability and the impact of control. In *Proceedings of ACM SIGCOMM 1999*, pages 301–313, 1999.

[15] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[16] S. Ben Fredj, T. Bonald, A. Proutiere, G. Regnie, and J. Roberts. Statistical bandwidth sharing: A study of congestion at flow level. In *Proceedings of ACM SIGCOMM 2001*, pages 111–122, 2001.

[17] P. L. George and H. Borouchaki. *Delaunay Triangulation and Meshing: Applications to Finite Element*. Hermis, Paris, 1998.

[18] Matthias Grossglauser and Jean-Chrysostome Bolot. On the relevance of long-range dependence in network traffic. *Transactions on Networking*, 7(5):629–40, October 1999.

[19] D.A. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[20] Guojun Kin, George Yang, Brian R. Crowley, and Deborah A. Agarwal. Network characterization server (NCS). In *Proceedings of the Tenth IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*. IEEE, August 2001.

[21] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *In Proceedings of USENIX Symposium on Internet Technologies and Systems*, 2001.

[22] Kevin Lai and Mary Baker. Measuring link bandwidths using a deterministic model of packet delay. In *Proceedings of ACM SIGCOMM 2000*, pages 283–294, 2000.

[23] C. Lawson. Transforming triangulations. *Discrete Mathematics*, 3:365–372, 1972.

[24] C. Lee and J. Stepanek. On future global grid communication performance. *10th IEEE Heterogeneous Computing Workshop*, May 2001.

[25] Bruce Lowekamp, David O'Hallaron, and Thomas Gross. Direct queries for discovering network resource properties in a distributed environment. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 38–46. IEEE Computer Society, August 1999.

[26] Bruce Lowekamp, David R. O'Hallaron, and Thomas Gross. Topology discovery for large ethernet networks. In *Proceedings of SIGCOMM 2001*. ACM, August 2001.

[27] Nancy Miller and Peter Steenkiste. Collecting network status information for network-aware applications. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[28] National Institute of Standards and Technology. NistNet. http://snad.ncsl.nist.gov/itg/nistnet, 2001.

[29] S. Owen. A survey of unstructured mesh generation. Technical report, ANSYS Inc., 2000.

[30] L. Rizzo. DummyNet, 1999. http://www.iet.unipi.it/~luigi/ip_dummynet.

[31] Jim Ruppert. A delaunay refinement algorithm for quality 2-dimensional mesh generation. *J. Algorithms*, 18(3):548–585, 1995.

[32] Aimin Sang and San qi Li. A predictability analsysis of network traffic. In *IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[33] Srinivasan Seshan, Mark Stemm, and Randy H. Katz. SPAND: Shared passing network performance discovery. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 135–46, December 1997.

[34] J. Shewchuk. *Delaunay refinement mesh generation.* PhD thesis, CMU, May 97.

[35] The Globus Team. The Globus Metacomputing Project. http://www.globus.org, 2001.

[36] Ajay Tirumala and Jim Ferguson. Iperf: The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, May 2001.

[37] David F. Watson. Computing the n-dimensional delaunay tessellation with application to voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.

[38] Rich Wolski. Dynamically forecasting network performance using the network weather service. Technical Report CS-96-494, UCSD, 1996.

[39] Rich Wolski, Neil Spring, and Chris Peterson. Implementing a performance forecasting system for metacomputing: The network weather service. In *Supercomputing '97*, 1997.

GRID PERFORMANCE AND RESOURCE MANAGEMENT USING MOBILE AGENTS

Beniamino Di Martino
Department of Information Engineering
Seconda Università di Napoli, Italy
beniamino.dimartino@unina.it

Omer F. Rana
Department of Computer Science
Cardiff University, Cardiff, UK
o.f.rana@cs.cf.ac.uk

Abstract

Mobile agents provide an important paradigm for supporting dynamic services in Computational Grids. We outline reasons why mobile agents are useful and how they can provide support for resource discovery and performance management in the context of service-oriented Grids. We also discuss factors which are likely to limit the uptake of the mobile agent approach, and how some of these restrictions can be overcome. This approach is subsequently exemplified by means of a mobile agent based programming and execution framework, the MAGDA system.

Keywords: mobile agents, Grid computing, resource management and discovery, performance management

1. Introduction and Previous Work

Increase in complexity of distributed systems requires improvements in current management schemes, and the need for paradigms beyond the client-server approach. This complexity can arise due to a number of factors, such as an increase in number of participating nodes, the software available on each node, the data dependencies between nodes or the performance differences observed across nodes. For instance, traditional methods for information processing require the data for an application to be moved from the data source to the point of processing. An approach gaining favour recently is the concept of moving the computation to the data, in the form of mobile code, where the code carries

some initialisation parameters, an executable, and current state to the remote machine. Once such code can be migrated across a network, it can be migrated subsequently to a number of different locations in an "itinerary", an approach referred to as a "mobile agent". This is particularly beneficial when dealing with large data sets, or when the exact sequence of service provision is not known in advance and is dependent on the order in which processing is performed. Hence, in a system involving mobile agents, the agent can visit a number of hosts, each of which offers part of the complete service required by the agent. The agent does not need to know the complete itinerary in advance, and may change its route based on information gathered at intermediate sites. This gives mobile agents the autonomy to make decisions based on interactions with local hosts as they move from one site to another, and negotiate services with hosts as they travel across a communication network. Furthermore, the mobile agent does not need a permanent connection to the originating host, making it ideal for tolerating temporary breakdowns in network connections, as on laptops and other mobile computing devices. However, each resource participating in the mobile agent system is required to provide a stationary agent (a secure "gateway" to the resources managed by the local system). In this paper, the use of mobile agents to support service execution in Grid environments is first outlined, followed by the description of the MAGDA system which may be used to realise these characteristics.

A framework or language is needed for developing mobile agent based systems. Such a language is needed to provide instructions for creating agents at a local site, initialising machines which can host mobile agents, storing/retrieving the state of an agent to/from disk, and activating/deactivating an agent etc. Various systems are currently available which support such instructions, some of which have gained popularity over the past few years. *TeleScript* from GeneralMagic was the first commercial transportable agent library [2], which has now been superseded by a Java based library called *Odyssey*. *Voyager* from ObjectSpace [4] is a Java class library for supporting distributed mobile components, and combines the Common Object Request Broker Architecture (CORBA) model with mobile Java agents. Voyager also supports a number of other CORBA services such as directory, object persistence and multi-cast services. *Aglets* from IBM Research [3] is also a popular Java class library allowing users to write mobile agents in a similar way to Applets, and it has recently become open source. Other mobile agent systems include *Agent Tcl* [1] which uses the scripting language Tcl to write executable code, *Sumatra* [5], Concordia [9] and Grasshopper [10]. The Grasshopper platform is particularly useful as it provides a means to develop mobile agents which can interact via standard service interfaces. It also provides various "add-ons", such as the MASIF [15] and FIPA [17] add-on, enabling agent interaction and management to be developed based on existing standards. The MASIF standard is

important to enable multiple mobile agents systems to co-exist. The standard addresses issues such as agent management, agent transfer, agent naming, and provisions for a location service. Another well known Java based system called *Mole* [8], developed at Stuttgart, allows *restricted migration*, where only the code and data for the agent is moved to a remote site, and not its execution state. The *D'Agents* system from Dartmouth [1, 6]provides support for multiple programming languages (Tcl, Java and Scheme), and also implements security mechanisms such as encryption, digital signatures and security modules which interact with resource managers to authenticate agents requesting to use system resources.

2. Service Support in Grid Computing

The dynamic provision of services forms an important aspect of Grid computing environments. Services may be provided by a number of different organisations and, except for a handful of core services (such as a certificate authority, for instance), are likely to be non-persistent. The "virtual organisation" [12] view of Grid environments especially requires the integration of such services, and current focus within the Open Grid Services Infrastructure (OGSI) is aimed at providing common encoding formats (in WSDL) and messaging standards (such as SOAP) for service interaction. Although the "service" paradigm enables the shift of emphasis on how a service is defined and how it is to be invoked, mapping the service reference to a particular resource (device) remains a vital step to realise a service-oriented Grid. Discovering physical resources which will host a particular service is therefore important to enable performance guarantees to be provided to a given user, and also to enable optimisation on how a particular service is to be executed subsequently.

Mobile agents are beneficial in offering and managing services which cannot be offered on a single resource. This could arise either as a result of a service being composed of multiple sub-services which cannot be physically executed at a given location, or due to overheads of data migration to a central repository. In both cases, a mobile agent could be assigned the task of migrating to the remote resources to undertake a particular part of the overall processing, and then return to the source with the result. We identify three particular roles that mobile agents could undertake in Grid environments:

- Support for *Resource Discovery*, where mobile agents interact with distributed registry services to locate resources of interest.

- Support for *Performance Monitoring*, whereby mobile agents interact with monitors on a given machine, or on a cluster, to gather performance attributes of a system.

- Support for *Code Migration*, whereby mobile agents can wrap analysis algorithms and migrate them to a remote location. This provides an alternative perspective on workflow, as the mobile agent interacts with various services to generate the final solution. The control flow is therefore equivalent to the migration of the mobile agent from one site to another.

2.1 Resource Discovery

The mobile agent paradigm can be useful to allow automated discovery of resources that form part of a Grid environment. This approach however necessitates the presence of a "stationary" agent at each site the mobile agent visits, and bootstrapping such a system may be a significant limitation in employing this approach. It may be difficult to convince system administrators at a remote site to install such stationary agents, especially as these provide a gateway to other system resources. However, this approach provides a significant alternative to the use of registry services, as currently provided with Grid Resource Information Service (GRIS) and Grid Index Information Service (GIIS) servers [7], which together constitute the MetaComputing (and Monitoring) Directory Service (MDS) in Globus, or UDDI based registries in Web Services. Another approach commonly found in Grid environments is the ClassAd mechanism in Condor [16], which provides a mechanism to undertake Matchmaking between resource requests generated by a user, and resource advertisements generated by a provider. When multiple suitable resources are found, a ranking mechanism is used to choose between the available resources. This approach is however limited to finding a single resource on which a user task can be executed, and does not provide any support for dealing with scenarios where multiple resources are required to execute a single task (or service). Identifying a set of resources (rather than an individual one) can also enable the organisation of these into a virtual machine, to better match resource demands for a particular application with resource provisions being made by a group of systems administrator. Liu et al. [18] propose an extension, as a *set-extended* ClassAd language, to enable resource aggregates to be specified, along with a set-matching mechanism that extends Condor Matchmaking. However, no particular matching criteria is provided within this framework, delegating the provision of this to a user instead. Liu et al. believe that the exact Matchmaking mechanism is so much domain dependent, that providing generic mechanisms is only of limited benefit. Furthermore, this approach also requires resource providers to advertise their capabilities, and resource users to publish their requirements to some central registry. The set-extended ClassAd language utilises a registry (the Resource Selector Service) to query the Globus MDS for resource information, and subsequently caches this in local memory for later use. Refreshes are based

on the expiry of a time-to-live parameter. Such registries are required to hold resource properties associated with a particular physical or logical domain, such as the machine architecture, the operating system supported, and limited details about software libraries, for all the hosts contained within that domain.

When utilising a mobile agent to interact with a registry service, only a single machine that has access to the registry service needs to run a stationary agent. A mobile agent visits a collection of such machines, and queries the registry to determine if resource properties are contained within the attached registry. Adverts generated by the resource provider therefore do not need to be accumulated at a single location, as the mobile agent can traverse domain boundaries to interrogate multiple registry services. The stationary agent is now responsible for scheduling queries on the local registry, and based on a query schema carried by the incoming mobile agent. Our query scheme is based on [14], and comprises of three main attributes of a resource encoded in an XML document: (1) capability/properties, (2) availability, and (3) ownership. Code segment 1 illustrates resource capabilities, outlining the properties that are captured by a mobile agent when it visits a site. A mobile agent can either perform a query for a single resource within the registry service managed by a stationary agent, or for all resources. The number of documents the mobile agent carries (as its local state) can therefore vary depending on the number of resources being requested by an application. Hence, attribute values in the XML document are either instantiated by querying the local registry, or by directly querying the local resource via the stationary agent. The use of XML, as a domain independent encoding format, is particularly useful to allow mobile agents to interact with stationary agents managed on different platforms. Hence, different parsing approaches for the XML documents may be available on particular hosts, and used to update parameters within the XML document.

```
<resource>
 <capability>
  <name value="emerald.cs.cf.ac.uk" short="emerald.cardiff">131.251.42.5</name>
  <type compound="true" value="0">C</type>
  <type compound="true" value="1">S</type>
  <operatingsys>Solaris7</operatingsys>
  <arch>SUN Ultra</arch>
  <loadavg type="DYNAMIC">0.332</loadavg>
  <idletime type="DYNAMIC" value="seconds">1442</idletime>
  <processcount type="DYNAMIC" value="NULL">120</processcount>
  <memfree type="DYNAMIC" value="MB">64</memfree>
  <memory type="cache" value="MB">4</memory>
  <memory type="RAM" value="MB">128</memory>
  <storage type="disk" value="GB">8</storage>
  <mmtimestamp>01.01.2000.15.15</mmtimestamp>
  <submittimestamp>02.01.2000.23.22</submittimestamp>
  <docversion>1.0</docversion>
 </capability>
```

Code Segment 1

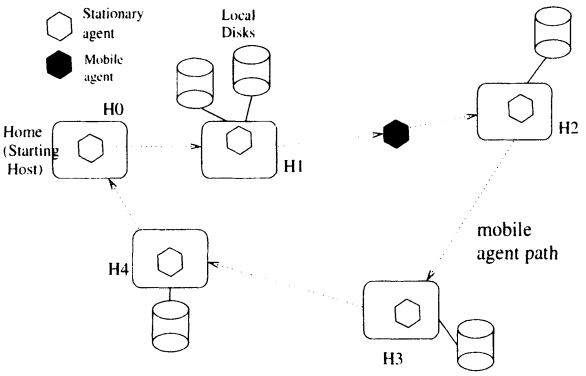


Figure 1. Mobile agent interacting with Grid registry services

The `<type>` tag indicates whether the resource can be used as a 'computation' (C) or a 'storage' (S) resource, via the Boolean `value` attribute. Each document also carries an `mmtimestamp` and a `version` number, to determine when the query was performed and which document structure was used. The time stamp and version number can be used by a system to detect whether the information carried by the mobile agent is valid. Figure 1 illustrates the systems architecture, indicating how the mobile agent gathers parameters from a particular site on its itinerary.

The mobile agent can also carry a Matchmaking algorithm, and can therefore provide an extension to the set-extended Matchmaking approach proposed by Liu et al. [18]. Hence, a site wishing to discover suitable resources will launch a mobile agent which carries an XML schema, and a Matchmaking algorithm, dependent on the particular demands of the application. Multiple concurrent mobile agents can access the same stationary agent, but select resources based on different match criteria. On completion, the mobile agent returns to the originating host, carrying with it references of the matches found. We assume that all hosts that can receive and generate mobile agents have been authenticated using a certificate authority, and do not envision the use deployment of mobile agents on arbitrary hosts. Security issues with receiving a mobile agent, and subsequently allowing the mobile agent to interact with local system resources remain open research objectives. ,

2.2 Performance Management

To support performance management in a Grid environment, mobile agents may be used to: (1) support load balancing across resources, (2) snap-shot the state of resources to discover the current state of a system. Load balancing with mobile agents is coarse-grained compared to other systems such as MPI. Each mobile agent now carries a self-contained application to a remote site, and can

delegate execution to slave agents. Both state-based (where parameters, such as workload, at a given site are used to determine the migration strategy), and model-based (where some predictive model is used to influence the migration strategy) have been demonstrated when using mobile agents, along with hybrid approaches which try to combine these [13]. The ability to migrate code automatically based on system state is a useful capability in Grid environments, as this allows for the more effective use of additional resources.

The load balancing strategy employed however relies on the ability to monitor the state of the system. This can be achieved using existing systems (such as NWS, or the "network health function" investigated in [11]), or via an event service managed by the stationary agent at each site. The mobile agent now visits each remote site, and takes a snap-shot of the system via the stationary agent. The mobile agent therefore does not carry any raw data, but only specific parameters that it wants to monitor on a given system. The capability to visit a number of sites along an itinerary, allows mobile agents to get an aggregate view of the resources that have been selected in the resource discovery phase. Access to the state of local resources can also be obtained via an SNMP-agent, providing this is supported on the local system. In this case, a mobile agent can issue an SNMP query via the local stationary agent, based on a common definition of resources defined in the SNMP Management Information Base (MIB). Using this approach, the output of an SNMP query is recorded, and the agent moves to another site along its itinerary. The use of SNMP is also beneficial as it provides a common naming scheme for network resources, facilitating the comparison of parameters across different machines (with different configurations). The XML document illustrated in code segment 1 may be extended with an additional set of attributes via a `<performance>` tag.

2.3 Provision of Mobile Services

Mobile agents can support the provision of "dynamic" workflows – especially when executing an aggregate service. The itinerary of the mobile agent determines the order in which each sub-service within the aggregate service (workflow) is to be executed. This is especially useful when each stage within the workflow does not require large data volumes to be communicate to a subsequent stage. The mobile agent therefore carries partial results (as its state) along its itinerary, finally returning results to the originating host on the completion of all sub-services. This mechanism provides a useful way to implement workflows where the the total number of stages are known, but not which particular sub-service implementation is to be invoked at a particular stage. The ability to dynamically determine which site to visit based on results obtained from the current site, enables a mobile agent to dynamically choose a location to support the execution of an aggregate service. Coupled with the discovery service

mentioned previously, this provides a powerful approach for service provision in an environment which changes often.

In this context, the mobile agent can also carry executable code (a "mobile service"), which acts as a trigger to launch a service via a stationary agent. Hence, interaction between a mobile agent and a stationary agent becomes an important concern – and may be supported through a Web Service (WSDL) based interface. In this case, each stationary agent will advertise locally managed services to the incoming mobile agent as WSDL documents, allowing a mobile agent to select a service of interest, and bind to it using a SOAP message. The stationary agent now provides an identical interface for access from a remote client or a mobile agent. The mobile-stationary agent interaction using SOAP is also useful to enable various network protocols to be supported by the stationary agent.

3. The MAGDA Framework

In this section we outline how the *MAGDA* (Mobile AGents Distributed Applications) toolkit [19] can be used to support some of the features discussed above. MAGDA is a framework for supporting the programming and execution of mobile agent, based on the Aglets workbench [3]. It supplements mobile agent technology with a set of features for supporting parallel programming on a dynamic heterogeneous distributed environment, as present in Grid systems. The (prototype) framework currently supports the following features:

- *collective communication* among mobile agents, with provision for a set of collective communication primitives;
- *dynamic workload balancing*, through centralised or distributed mechanisms;
- *dynamic system monitoring* for each node of the executing platform;
- *automated mechanisms for agent migration and cloning*, driven by policies based on system monitoring, in terms of utilisation, idle time, memory usage and different run-time events (like shutdown);
- *agent authentication*, to support security critical applications;
- *remote agent creation*, which provides the possibility to activate an agent from different platforms, including wireless hand-held devices;
- a *skeleton-based* parallel programming environment, based on specialisation of *Skeleton* Java interfaces and classes;
- *integration of MA programming paradigm and OpenMP*, for programming hierarchical distributed-shared memory multiprocessor architec-

tures, in particular heterogeneous clusters of SMPs (and uniprocessor) nodes.

3.1 Performance management through dynamic workload balancing and system monitoring

As outlined before, in order to support performance management in a Grid environment, mobile agents can be used to support load balancing across resources, and to monitor the state of resources to discover the current state of a system.

MAGDA utilises a *coordinator agent* to support and control load balancing, which may be adapted for an arbitrary service. The coordinator agent communicates, by message passing, only with an Aglet superclass. The coordinator manages a list of registered workers and a list of available free hosts. When the execution of the user agent is started, a coordinator is created, to poll the working agents registered with it. This is undertaken to determine the state of computation (defined as the percentage of computation performed with respect to the computation assigned to each) of each working agent. The registered agent's references are stored in a vector, ordered according to their computation state; the ordering of this vector thus represents the relative speed of each worker with respect to the others. It also gives a representation of the state of the computation. A load unbalance event occurs when:

1 a worker ends the computation assigned to it, and becomes idle;

2 the slowest worker has completed a percentage of the computation amount assigned to it which is far below the percentage completed by the fastest worker (determined by a fixed threshold on the difference of the two percentages).

In the first case, the idle worker notifies the coordinator of its availability to receive workload; the coordinator then asks the slowest worker (according to the vector of computation states) to send a portion of its workload to the idle worker. In the second case, the following two situations can occur:

1 the list of available hosts is not empty: the coordinator then creates a generic slave agent and dispatches it to the target machine, then asks the slowest worker to send a part of its load to the slave.

2 the list of available hosts is empty: the coordinator then asks the slowest worker to send part of its load to the fastest worker.

The coordinator periodically updates the vector of the computation states, by polling at fixed or variable frequency, or using user-defined criteria.

In MAGDA, exchange of workload between agents is equivalent to the exchange of serializable objects belonging to a class defined by the user (`UserWorkload`), implementing the the `Load` interface. The data declared in class `UserWorkload` represent the tasks to be undertaken and their properties, while the method `exec` of the `Load` interface identifies the data. Such a method is invoked by the receiver to perform the computation. The user can define different kinds of `UserWorkload` classes, according to the kind of computational tasks to send (in different points of the program we can treat different problems) or according to the destination agent (slave needs all data and code, while worker can reuse its code or its data). A user in MAGDA is responsible for the following activities:

- to calculate and communicate the percentage of work done;
- to create and prepare the load assigned to the fast/idle worker or to the slave;
- to specify if the fast/idle worker or the slave must return the result of its computation;
- to elaborate the results eventually received from the fast/idle worker or the slave.

To personalise the behaviour of the coordinator the user must provide the following information:

- the frequency rate of polling, for updating the vector of computation states;
- a threshold for triggering load transfer between workers
- all the URLs of the hosts of the cluster.

To personalise the behaviour of the workers the user can set some variables inherited from the superclass. In particular, the user can choose:

- if a worker wants to receive any load from the others;
- if it needs to be given back any result possibly computed by the workload passed to another worker agent.

The MAGDA framework provides primitives which allow dynamic monitoring of system parameters such as the CPU and memory utilisation, and network parameters such as bandwidth (particularly useful when transferring mobile agents). A set of Java APIs have been developed to periodically collect these parameter values and are available for the both Linux and Solaris. The

MAGDA server uses the Java runtime system to make operating systems calls which help to estimate and record system status, which may be subsequently queried by each agent. The use of a monitoring thread to gather activity and resource usage allows a programmer to design strategies to optimise system utilisation and throughput. As indicated previously, in order to gauge global system activity, a dedicated travelling agent can visit servers of the cluster and collect all the local system information. The collected information can be used to optimise the distribution of the application workload among the agents or to move an agent from a heavier loaded computing machine to a less loaded one.

3.2 Provision of Workflow mobile services through skeleton based mobile agent programming, collective coordination and security mechanisms

Utilising the mobile agent paradigm in Grid computing is a non-trivial task because the mobility feature introduces additional difficulties in designing coordination, synchronisation and communications among the different tasks. In most cases, the use of algorithmic skeletons can ease the programming task, and also the subsequent mapping on parallel systems. MAGDA provides a set of Java packages, which enable the development of distributed applications by adopting a 'skeletons'-like approach, exploiting the peculiar features of both Object Oriented and Mobile Agents programming models [20]. By means of the provided skeleton interfaces the programmer is able to implement an application by specialising an existing structure, and utilising the set of functionalities that the mobile agents framework offers.

Selected workflow patterns can be suitably represented through predefined skeletons, and concrete workflow mobile services can be implemented by instantiating such skeletons. As an example, a number of workflow patterns can be represented by the *Processor Farm* and the *Divide and Conquer* skeletons. Those skeletons have been realised in MAGDA and experimented [20].

In *Processor Farm* skeleton the master process create a number of slaves and assigns some work to everyone of them. The slaves compute their work and return the results to the master. *Task Queue* is the most general *Farm*-like skeleton, every slave may produce new work to be performed by itself or by other slaves. The second algorithmic skeleton belongs to *Divide and Conquer*-like skeleton class, but not to the highest abstraction level. It is an example of *Tree computation* algorithmic skeleton. It solves the initial problem dividing it in several subproblems assigned to different agent workers. The data flow from the root into the leaves and the solutions flow back up towards the root.

As indicated previously, a mobile agent should be able to locate/discover stationary agents (active servers), and to communicate with other agents without knowing their geographical location. In a dynamic environment, such as

the Computational Grid, mechanisms for collective communication and synchronisation become essential. MAGDA provides collective communication primitives such as broadcast and multicast, and collective synchronisation operations through a regular binary tree topology structure, which is built and managed by the active servers through a suitable protocol.

Security remains a crucial issue when deploying mobile agents within Grid environments, especially when distributing mobile code. Approaches in the Aglets workbench (on which MAGDA is based) are limited, allowing the administrator to design security policy in order to grant or deny to the agents some services such as dispatching, cloning, file system reading or writing, socket opening etc. A user may also sign JAR (Java Archives) files, but little else exists. Such security functions are limited to the agent code and not to the value of its data at the dispatching and recovering time. Also, server authentication prior to the transmission of a mobile agent to a remote site is also not available. MAGDA provides a Java API to sign code and data of an agent application. A programmer can now choose to use digital signature to authenticate their classes and the current state of an agent, prior to dispatching the code. The hosting server can check the integrity of received bytes and authenticate them. The server can be initialised to store the received signed code in order to grant itself from senders repudiation.

4. Conclusion

Mobile agents provide an important paradigm for supporting Grid services, in three particular ways: (1) support for mobile service, (2) support for performance monitoring and load balancing, and (3) support for resource discovery. Techniques for utilising mobile agents in each of these domains are first outlined, followed by the discussion of the MAGDA mobile agent toolkit to support these features. Emerging interest in Grid Services, based on WSDL and SOAP provide an important first step to standardise service access and descriptions in Grid systems. Mobile agents will provide a useful next step in utilising these services more effectively, especially where the complexity of a system restricts the use of existing client/server paradigms.

References

[1] R. Gray, "Agent Tcl: A transportable agent system", *Proceedings: 4^{th} annual Tcl/Tk Workshop, Monterey, Ca*, 1996.

[2] J. White, "Mobile Agents White Paper", *General Magic Inc.*, 1996.

[3] IBM Research, "The Aglets Software Development Kit", see Web site at: http://www.trl.ibm.co.jp/aglets, 1998.

[4] ObjectSpace, "Voyager: A Java based Mobile Object System", see Web site at: http://www.objectspace.com/, 1998.

[5] M. Ranganathan, A. Acharya, S. Sharma, and J. Saltz, "Network-aware Mobile Programs", *Proceedings: USENIX Technical Conference*, pages 91–104, 1997.

[6] D. Rus, R. Gray, and D. Kotz, "Transportable Information Agents", *Journal of Intelligent Information Systems*, 9:215–238, 1997.

[7] K. Czajkowski, S. Fitzgerald, I. Foster, C. Kesselman, "Grid Information Services for Distributed Resource Sharing", *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing* (HPDC-10), August 2001.

[8] M. Stasser, J. Baumann, and F. Hohl, "Mole, A Java based Mobile Agent System" *2nd ECOOP Workshop: Mobile Object Systems*, 1996.

[9] R. Koblick, Concordia, *Communication of the ACM*, 42(3), March 1999.

[10] IKV++ Technologies AG, The Grasshopper Platform, see Web site at: http://www.grasshopper.de/, 2002.

[11] O. Tomarchio, L. Vita, and A. Puliafito, Active Monitoring in Grid Environments using Mobile Agent Technology, in 2^{nd} *Workshop on Active Middleware Services (AMS)* at HPDC-9, Pittsburg, USA, 2000.

[12] I. Foster, C. Kesselman, J. Nick, S. Tuecke, The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, at Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002. Available at: http://www.globus.org/research/papers.html, 2002

[13] C. Georgousopolous and O.F. Rana, Combining state and model based approaches for mobile-agent load balancing, *Proceedings of ACM Symposium on Applied Computing*, Florida, March 2003 (to appear)

[14] O.F. Rana, D. Bunford-Jones, D.W. Walker, M. Addis, M. Surridge, and K. Hawick, Resource Discovery for Dynamic Clusters in Computational Grids, *Procedings of Heterogeneous Computing Workshop*, at IPPS/SPDP, San Francisco, California, April 2001.

[15] OMG, The Mobile Agent Systems Interoperability Facility (MASIF) Standard, see Web site at: http://www.fokus.gmd.de/research/cc/ecco/masif/, 1998.

[16] R. Raman, M. Livny, M. Solomon, Matchmaking: Distributed Resource Management for High Throughput Computing, in *Proceedings of IEEE Symposium on High Performance Distributed Computing*, 1998.

[17] FIPA, Specification for Intelligent Agents, see Web site at: http://www.fipa.org/, 1997.

[18] C. Liu, L. Yang, I. Foster, and D. Angulo, Design and Evaluation of a Resource Selection Framework for Grid Applications, *Proceedings of IEEE International Symposium on High Performance Distributed Computing* (HPDC-11), Edinburgh, UK, July 2002

[19] R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque, MAGDA: a software environment for Mobile AGent based Distributed Applications, Proc. of 11^{th} *Int. Workshop on Parallel and Distributed Processing*, Feb. 2003, Genova, Italy (to appear).

[20] R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque, Mobile Agent Programming for Clusters with Parallel Skeletons, Proc. of Int. Conf. *VECPAR 2002*, June 26-28 2002, Porto, Portugal.

MONITORING OF INTERACTIVE GRID APPLICATIONS*

Bartosz Baliś, Marian Bubak, Włodzimierz Funika
Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
{balis,bubak,funika}@uci.agh.edu.pl

Tomasz Szepieniec
Academic Computer Center – CYFRONET-AGH, Nawojki 11, 30-950 Kraków, Poland
t.szepieniec@cyfronet.krakow.pl

Roland Wismüller
LRR-TUM – Technische Universität München, D-80290 München, Germany
and
Institute for Software Sciences, University of Vienna, A-1090, Wien Austria
wismuell@in.tum.de

Abstract This paper presents the OCM-G, a Grid application monitoring system. The OCM-G is aimed to provide services via which tools supporting application development are enabled to gather information, manipulate, and detect events that occur when applications are running. The functionality of the OCM-G is available via a standardized interface, On-line Monitoring Interface Specification (OMIS). The OCM-G is designed to work in a Grid environment. This implies a distributed and decentralized design which allows for a large-scale scalability and capability to handle multiple applications, users and tools at the same time, while ensuring security. The design of the OCM-G assumes that one part of it is permanent which allows it to work as a Grid service and additionally enables communication through firewalls, whereas another part is transient and private to each Grid user, what solves the major security problems. In the paper, we provide a short overview of OMIS, describe the design of the OCM-G and discuss Grid-specific requirements including necessary OMIS extensions as well as security issues.

*This work was partly funded by the European Commission, project IST-2001-32243, CrossGrid

Keywords: Grid computing, on-line monitoring, services, performance analysis, performance tools

1. Introduction

Grid computing has become one of the promising ways of solving data and compute intensive problems. One of its most challenging aspects is the necessity for software support of the optimization of development and use of the vast Grid opportunities what may be realized with monitoring facilities.

The subject of this paper is an application monitoring system called OCM-G – Grid-enabled OMIS Compliant Monitoring System. The OCM-G is meant to enable investigation and manipulation of parallel distributed applications running on the Grid [9–11] and to provide a basis for building tools supporting development of parallel applications for the Grid.

This research is part of the EU CrossGrid project [7]. The applications of the CrossGrid project feature interactions with a human in a processing loop, being compute- and data-intensive [6]. To facilitate development of these applications, the project develops a tool, named G-PM, which not only allows to measure just the standard performance metrics, but enables to determine higher-level performance properties and application specific metrics [5]. The above features motivate building a specific monitoring infrastructure [2] as an autonomous, distributed system which offers monitoring services via a standardized interface – OMIS (On-line Monitoring Interface Specification).

In the following we focus on our approach to Grid monitoring, provide a short overview of OMIS, describe the design of the OCM-G, and discuss Grid-specific extensions to OMIS and security issues.

2. Monitoring of Grid Applications

A comparative survey of grid-oriented monitoring tools is given in [1]. The Globus Heartbeat Monitor [13] supports monitoring the state of processes. Network Weather Service [19] monitors and forecasts the performance of networks and computational resources. The Information Power Grid project is developing its own monitoring infrastructure [18] built of sensors, actuators, and event services. NetLogger [12] is a tool for analysis of communication inside applications. GRM and PROVE [15] are developed for performance monitoring of applications running in distributed heterogeneous systems. Autopilot/Virtue [17, 16] is a performance measurement and resource control system. The DataGrid project [8] is developing the Relational Grid Monitoring Architecture oriented towards applications which operate on large data files.

None of the mentioned tools supports on-line monitoring of applications with high scalability and efficiency which are crucial for large interactive applications. Also, none of the described approaches is based on an autonomous

monitoring system accessible via a standard interface which allows to use it as a Grid service on which newly developed tools may be based. A system with properties that are similar to those of the OCM-G is GRM/PROVE. The latest version of the GRM monitor is integrated with R-GMA which uses permanent producers and this means that for each type of information we are interested in a new producer has to be created. Moreover, GRM does not allow for manipulations on the target system objects.

Our approach to Grid monitoring is *application oriented* – it provides information about running applications in the on-line mode. The information is collected by means of selective instrumentation at run-time. "Selective" means that instrumentation can be activated or deactivated on demand, what ensures a minimal extent of monitoring overhead since we may focus on the data we are interested in at the given moment. The OCM-G is designed as an autonomous system accessible via a standardized interface – OMIS (On-line Monitoring Interface Specification) [14]. Thus, newly developed tools compliant with OMIS are immediately enabled to use the monitoring facilities provided by the OCM-G.

Our experience in application monitoring includes the development of a tool environment based on the OCM – an OMIS-compliant Monitoring System for clusters. The practice proved that OMIS enables to adapt existing tools to new programming paradigms, e.g. PVM and MPI, with a minimal coding effort [4].

3. A Short Overview of OMIS

In the OMIS approach the target system forms a hierarchy of objects. In a cluster environment, the top-level objects are *nodes* which contain *processes* composed of *threads*. The objects are identified by so called *tokens*, e.g. `n_1`, `p_5`, `t_10`, etc.

OMIS defines three types of services: *information* services to obtain information about objects, *manipulation* services to manipulate objects, and *event* services to detect events in the target system (especially inside applications). The information and manipulation services are also called *actions*. Examples of the services are: `proc_get_info` (return information about a process), `thread_stop` (stop a thread/process), `thread_has_started_libcall` (thread/process has started an invocation of a function call).

By combination of services two types of *monitoring requests* may be formed: *unconditional*, which comprises one or more actions, and *conditional*, which is composed of one event service and one or more actions. For unconditional requests, the actions are executed immediately and only once, whereas in case of conditional requests the actions are executed whenever the associated event occurs; this may take place multiple times. Below there are two examples of monitoring requests:

- `:proc_get_info([p_1], 3)`

  (return information on process `p_1`, where '3' is a bitmask which specifies the type of information to be returned),

- `thread_creates_proc([p_1]): proc_get_info([$newproc)], 3)`

  (if process `p_1` creates a new process, returns information on the newly created process).

4. OMIS extensions for the Grid

The adaptation of OMIS to the Grid required several extensions to the specification. First of all, the object hierarchy was extended by new types of objects – *sites* – which are on the top of the hierarchy. It reflects the structure of the Grid which can be viewed as a collection of sites composed of individual nodes hosting processes. OMIS must also be extended by the following new services:

- services related to the new *site* objects (e.g., to get information about a site);
- services for infrastructure-related metrics which are not necessary in cluster environments (e.g., return information about a network connection);
- services for handling multiple applications (return the list of applications, return the list of processes of an application, etc.);
- other services which add a novel functionality, such as services for handling *probes* – objects inserted by the user into the source code to define arbitrary events and user-defined metrics for performance analysis.

5. Design of the Grid Application Monitoring System

Due to a Grid-wide scale, the OCM-G has to be properly distributed and decentralized to address high scalability and efficiency demands – applications can potentially be composed of a large numer of processes and distributed across multiple sites.

The OCM-G comprises two types of components: Local Monitors and Service Managers.

- Service Managers (SM) are the permanent part of the OCM-G and there is one SM per site of the Grid. Each SM exposes the monitoring functionality – tools which want to carry out a monitoring session contact the nearest (i.e. of the same site) SM to which they submit OMIS requests and from which they receive replies. An SM distributes OMIS requests

to Local Monitors, if the target object is located on the same site and/or to the other SMs to reach objects located outside the site.

- Local Monitors (LMs) are transient components in the OCM-G; these are created on each host of the Grid where application processes to be monitored exist. An LM executes OMIS requests accepted from a SM and sends back the replies. The execution of requests includes performing requested actions on application processes or nodes, and also managing the capturing of events.

The communication scheme in the OCM-G is designed in such a way that a LM is never contacted from within remote sites. This approach should be acceptable for site administrators who will have to open only one additional communication channel to the outside world on only one host of the site for the SM. Moreover, if sites are behind firewalls, this can be the only solution which enables communication via firewalls. Since the SMs are permanent, they can be assigned a well-known port which will be open on firewalls.

The architecture of the OCM-G is shown in Fig. 1. Note that information about each application does not need to be distributed among all Service Managers. One application usually involves only part of the OCM-G (i.e., part of all Service Managers). This part is a Virtual Monitoring System (VMS) for that application (gray part in Fig. 1). One of the Service Managers can be designated to be the Main Service Managers (MainSM) for a given VMS. The MainSM is always supposed to keep up-to-date information about the application and be first notified about all new events (e.g. creation, destruction or migration of a process). The location of the MainSM for a given application can be accessible via an external Grid information service. Thus, for instance, if a new application process is created on a site on which no other processes of this application exist, this site's SM is first notified about the creation. Next,

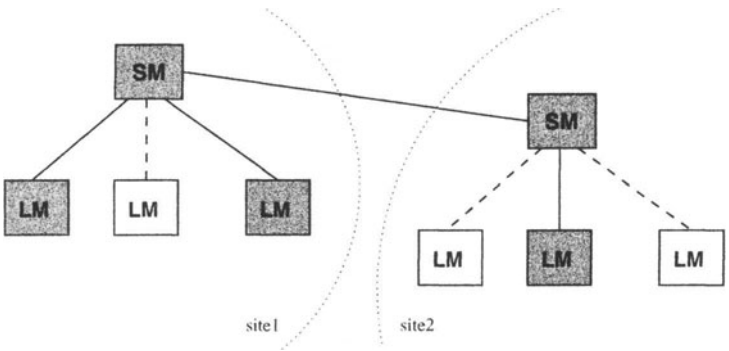


Figure 1. A virtual monitoring system based on the OCM-G architecture

it refers to the information service for the location of the MainSM, and finally notifies it about the event, and becomes a new member of the VMS.

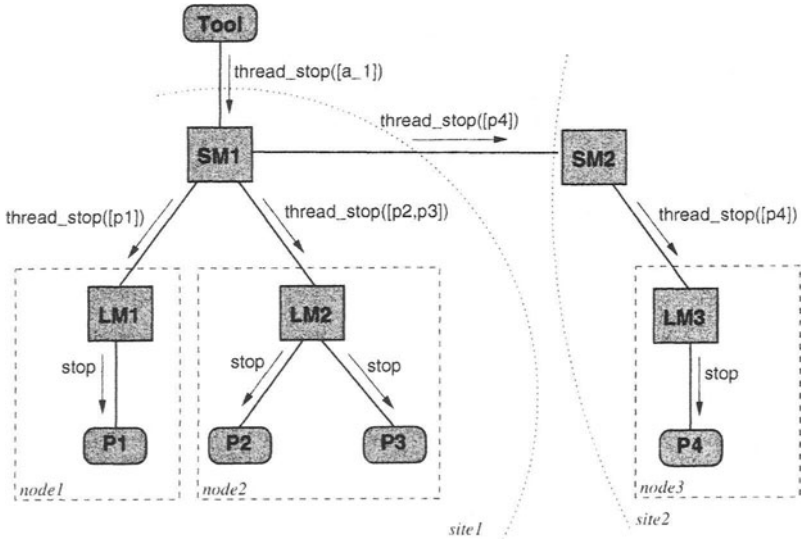


Figure 2. Distribution of an OMIS request

Fig. 2 shows an example of distribution of an OMIS request, `thread_stop` which is used to stop processes. The argument given to this service is a list of processes to be stopped, in this case it is `a_1` which identifies the whole application and is expanded by the Service Manager to the list of all the processes of the application. Since the four processes of the application are distributed across two different sites and two nodes on the local site, the SM splits the request into subrequests and forwards each one to the appropriate LMs and to the SM on the remote site which in turn forwards the message to the appropriate LM (LM3). Each LM executes the request – it stops the processes listed in the request. To enable the above operations, the actual knowledge about localization of application processes must be accessible, so a mechanism to keep it up-to-date within the OCM-G is needed.

A tool can connect to the monitoring system via the nearest Service Manager. Fig. 3 presents a simplified state chart diagram for a tool. At the beginning, the

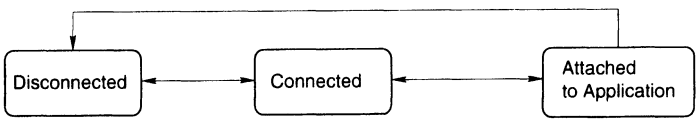


Figure 3. State transitions of a tool in the start-up phase

tool is in the *disconnected* state. After the tool *connects* the OCM-G, it may get information about all running applications. Then one of the applications can be chosen and the tool should *attach* to the application of choice. Only upon having attached to the application the tool can monitor it. It should be noted that we can attach to a running application and begin monitoring at any time provided that the application was properly prepared to enable monitoring.

A single application can be monitored not only by one tool, but we may use multiple tools concurrently to monitor different aspects of the application (e.g., measure performance with a performance analysis tool and at the same time provide some additional visualization with another tool). This *interoperability* is only possible due to the autonomous monitoring infrastructure of the OCM-G. An example of interoperability of tools in a OCM-based monitoring environment on clusters is described in [3].

6. Security Issues

Due to the fact that the OCM-G is a global Grid service, handling multiple applications, users, and tools implies several security problems. Firstly, because Local Monitors are allowed to manipulate processes; secondly, because Service Managers handle the requests of multiple users. On both levels authentication mechanisms are needed: at the level of LM to prevent malicious users from manipulating other users' processes, and at the level of SM to ensure that each request is sent by an authorized user.

To resolve the first problem, Local Monitors are started as user processes. This means that we have to start an LM for each host and each user, so the security on this level is then ensured directly by an operating system.

We cannot do the same with Service Managers, since they are permanent and well-known. Fortunately, SMs don't do any manipulations to the target system, so they need not be priviledged processes. This, however, does not solve the problem of issuing unauthorized requests. To resolve this, Grid certificates will be incorporated – each request sent to a SM should be signed and before it will be forwarded to Local Monitors to be checked whether the user who issued the request is the owner of the application on which the request is intended to be performed.

Although the authentication of each request ensures a good level of security also at the level of LMs since they accept only properly signed requests, it is still safer and probably necessary to run LMs as user processes to decrease the possibility of a security hole and protect from accidental (or intentional) damage done by users who have Grid certificates and are allowed to monitor applications. Furthermore, system administrators may be not willing to allow for a daemon with root priviledges anyway.

7. Summary

We have presented the OCM-G system as a facility for monitoring applications on the Grid. The OCM-G is designed as a Grid Service – it is permanent, being accessible via a well-defined interface, OMIS. The architecture of the monitoring system ensures a high scalability and efficiency of application monitoring. We described solutions which allow secure monitoring multiple applications distributed across multiple sites owned by multiple users and monitored by multiple tools.

At present we are working on the first prototype implementation of the OCM-G which will be ready by February 2003. The first implementation will support all services defined by the OMIS 2.0 specification and some new Grid extensions needed for the first prototype. This version will run on one site only, supporting one application and one tool. A fully functional version of the OCM-G will be available at the end of the CrossGrid project.

References

[1] Balaton, Z., Kacsuk, P., Podhorszki, N., and Vajda, F.: Comparison of Representative Grid Monitoring Tools.
http://hepunx.rl.ac.uk/edg/wp3/meetings/budapest-jan01/NP%20Grid%20Tools.pdf

[2] Balis, B., Bubak M., Funika W., Szepieniec, T., and Wismüller R.: An Infrastructure for Grid Application Monitoring. In: D. Kranzlmüller, P. Kacsuk, J. Dongarra, J. Volker (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 9th Eurpean PVM/MPI Users' Group Meeting, Linz, Austria, September/October 2002, LNCS 2474, pp. 41-49, Springer 2002

[3] Bubak, M., Funika, W., Baliś, B., and Wismüller, R: Interoperability of OCM-based On-line Tools. In J. Dongarra, P. Kacsuk, and N. Podhorszki (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 7th European PVM/MPI Users' Group Meeting, Balatonfüred, Hungary, September 2000, LNCS 1908, pp. 242–249, Springer, 2000

[4] Bubak, M., Funika, W., Baliś, B., and Wismüller, R.: On-line OCM-based Tool Support for Parallel Applications. In: Yuen Chung Kwong (ed.): Annual Review of Scalable Computing, 3, Chapter 3, 2001, Singapore

[5] Bubak, M., Funika, W., and Wismüller, R.: The CrossGrid Performance Analysis Tool for Interactive Grid Applications. In: D. Kranzlmüller, P. Kacsuk, J. Dongarra, J. Volker (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 9th Eurpean PVM/MPI Users' Group Meeting, Linz, Austria, September/October 2002, LNCS 2474, pp. 50-60, Springer 2002

[6] Bubak, M., Malawski, M., and Zajac, K.: Towards the CrossGrid Architecture. In: D. Kranzlmüller, P. Kacsuk, J. Dongarra, J. Volker (Eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 9th Eurpean PVM/MPI Users' Group Meeting, Linz, Austria, September/October 2002, LNCS 2474, pp. 16-24, Springer 2002

[7] CrossGrid Project: http://www.eu-crossgrid.org

[8] DataGrid Project: http://www.eu-datagrid.org

[9] Foster, I., Kesselman, C. (eds.): The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, 1999

[10] Foster, I., Kesselman, C., Tuecke, S. The Anatomy of the Grid. Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications, 15(3):200-222, 2001
http://www.globus.org/research/papers/anatomy.pdf

[11] Foster, I., Kesselman, C., Nick, J.M., and Tuecke, S.: The Physiology of the Grid. An Open Grid Services Architecture for Distributed Systems Integration, January 2002
http://www.gridforum.org/ogsi-wg/drafts/ogsa_draft2.9_2002-06-22.pdf

[12] Gunter, D. et al.: NetLogger: A Toolkit for Distributed System Performance Analysis. Proc. IEEE Mascots 2000 Conference, Lawrence Berkeley Natl. Lab., Report LBNL-46269

[13] Heartbeat Monitor Specification. http://www.globus.org/hbm/heartbeat_spec.html

[14] Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, 1997
http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz/OMIS/Version-2.0/version-2.0.ps.gz

[15] Podhorski, N., Kacsuk, P.: Design and Implementation of a Distributed Monitor for Semi-on-line Monitoring of VisualMP Applications. Proc. DAPSYS 2000, Balatonfured, Hungary, pp. 23-32, 2000

[16] Shaffer, E. et al.: Virtue: Immersive Performance Visualization of Parallel and Distributed Applications. IEEE Computer, 44-51, Dec. 1999

[17] Vetter, J.S., and Reed, D.A.: Real-time Monitoring, Adaptive Control and Interactive Steering of Computational Grids. The International Journal of High Performance Computing Applications, 14:357-366, 2000

[18] Waheed, A., et al.: An Infrastructure for Monitoring and Management in Computational Grids. Proc. 5th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, March 2000

[19] Wolski, R., Spring, N., and Hayes, J. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Future Generation Computer Systems, 15:757-768, 1999

THE UNICORE GRID AND ITS OPTIONS FOR PERFORMANCE ANALYSIS

Sven Haubold, Hartmut Mix, Wolfgang E. Nagel
Center for High Performance Computing (ZHR)
Technical University Dresden, Dresden, Germany
{haubold,mix,nagel}@zhr.tu-dresden.de

Mathilde Romberg
Central Institute for Applied Mathematics (ZAM)
Research Center Juelich, Juelich, Germany
m.romberg@fz-juelich.de

Abstract UNICORE (Uniform Interface to Computer Resources) is a software infrastructure to support seamless and secure access to distributed resources. It has been developed by the projects UNICORE and UNICORE Plus in 1997 - 2002 (funded by the German Ministry of Education and Research) and is going to be enhanced in the EU-funded projects EUROGRID and GRIP. The UNICORE system allows uniform access to different hardware and software platforms as well as different organizational environments. The core part is the abstract job model. The abstract job specification is translated into a concrete batch job for the target system. Besides others, application specific support is a major feature of the system. By exploiting the plugin mechanism, support for performance analysis of Grid applications can be added. As an example, support of Vampirtrace has been integrated. The UNICORE user interface then gives the option to add a task using Vampirtrace with runtime configuration support into a UNICORE job and retrieve the generated trace files for local visualization. Together with the support for compilation and linkage and for metacomputing, the plugin mechanism may be used to integrate other performance analysis tools in future.

Keywords: Grid computing, UNICORE, Vampir, performance analysis, application specific interface, Java

1. Introduction

The Grid [4] has become the major research topic for all kinds of scientists. Computer scientists, physicists, biologist, engineers, and other application people try to get their applications within this new computational infrastructure to real production mode. It is still a very complex environment, and the only way to make significant steps is collaboration across all disciplines. Actually, in many projects the solution of a special application problem takes the major amount of work, others focus on the definition of new functionality or features which are needed to provide a flexible environment. The process of standardization - beyond de-facto standards - has fruitfully started, and now the aspects of performance get into focus.

Performance analysis in a Grid environment has several aspects: Performance of distributed applications, performance of Grid components, and performance of the overall system. Projects like for example crossgrid [16] work on verification and performance prediction tools as well as detection of performance bottlenecks in applications in Grid environments. Crossgrid will propose a set of performance metrics to describe concisely the performance capacity of Grid configurations and application performance, and it will develop and implement benchmarks that are representative of typical Grid workloads. Based on that, it will provide on-line tools which allow application developers to measure, evaluate, and visualize the performance of Grid applications with respect to data transfer, synchronization, and I/O delay as well as CPU, network, and storage utilization.

The Global Grid Forum [17] works on the standardization in the field of monitoring and performance analysis. Working Groups on Discovery and Monitoring Event Description (DAMED-WG) and Network Measurement (NM-WG) together with the Research Group Grid Benchmarking (GB-RG) deal with the definition of a basic set of monitoring event descriptions, the definition of a hierarchy of network measurements for Grid applications and services, and the definition of metrics to measure performance of Grid applications and architectures and rate functionality and efficiency of Grid architectures.

This article will focus on the flexible support for application performance analysis in the UNICORE framework. UNICORE is a major initiative to provide a stable Grid production framework for HPC applications running on different sites in Germany and beyond. It provides an interface which allows to transparently use computer resources in different Grid centers. We will describe the mechanisms which have been developed to support an easy, flexible, and transparent performance analysis process for applications running on different sites. As the basis, we have used Vampir as the well established performance analysis tool for parallel environments.

The remainder of this article first gives a general introduction into the UNICORE Grid infrastructure, the user functions it offers, and relevant interfaces. A description of Vampir and its capabilities follows in Section 3 while Section 4 explains the integration of Vampir into the UNICORE framework. The outlook summarizes the exploited features and describes options for further performance analysis.

2. UNICORE

The idea to build a *uniform interface to computing resources* goes back to mid 1996 when the management of the supercomputer centers in Germany were looking for improved end user access and closer cooperation: Already at that time the vision was that the users should be able to use the most appropriate computer system for their application without taking care about system and site specific commands, file spaces, security mechanisms, and such. This initiative has been supported and funded in part by the BMBF (German Ministry for Education and Research) in two projects, UNICORE (1997-1999, [18]) and UNICORE Plus (2000-2002, [19]). The project goal has been to provide a seamless, secure, and intuitive interface to distributed heterogeneous computer resources. Partners in these projects are from universities, research laboratories, and software companies[1]. This group has developed the Grid system UNICORE described in more detail in the next subsections.

2.1 The Grid Infrastructure UNICORE

The UNICORE three tier architecture developed consists of user, server, and target system tier as shown in Figure 1 (see also [12]).

The user tier comprises the graphical user interface. It offers the functions to prepare and control UNICORE jobs and to set up and maintain the user's security environment. The security architecture is based on the Secure Socket Layer (SSL) protocol. SSL is used for all communication over public networks. Communication over the intra-net at a site is done unencrypted by default but SSL is a configurable option. From the user input, the user interface generates an Abstract Job Object (AJO) which is sent via SSL to the Gateway. The AJO is a key component in the architecture. It comprises the UNICORE protocol between user interface and the Network Job Supervisor together with the abstract job specification generated from the user input (references [3] and [11] explain

[1] Partners are: Forschungszentrum Juelich GmbH, Deutscher Wetterdienst (DWD), Pallas GmbH, Rechenzentrum der Universitaet Stuttgart (RUS), Konrad Zuse Zentrum fuer Informationstechnik (ZIB), Leibniz Rechenzentrum der Bayrischen Akademie der Wissenschaften (LRZ), Paderborn Center for Parallel Computing (PC^2), Rechenzentrum der Universitaet Karlsruhe (RUKA), Fujitsu Laboratories of Europe (former fecit), Genias Software GmbH (UNICORE project only), and Zentrum fuer Hochleistungsrechnen an der TU Dresden (ZHR, UNICORE Plus project only).

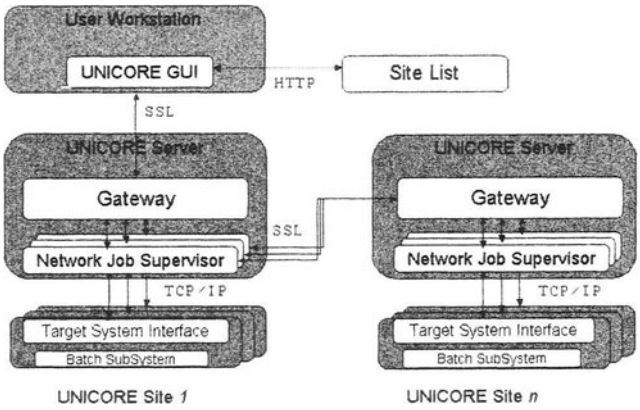


Figure 1. Architecture Overview

the AJO in more detail). The Gateway is the entrance to a site. It takes care of user authentication, secure communication between client and server, and provides the client with information about the available resources at a site. It also talks to the Network Job Supervisor (NJS) servers at its site to send jobs and data, status requests, and control commands for further processing and to receive data to make it available to the user.

Each Network Job Supervisor controls one Vsite (Virtual Site) which is a single system or one cluster of systems governed by one resource management system and sharing the same userids and file space. It fulfills the following tasks:

- analyze the AJO (representing the UNICORE Job) to extract local and remote jobs and tasks;
- map the UNICORE userid to the local userid for the target system (Vsite);
- translate the local tasks contained in the AJO into real batch jobs for the target system;
- send sub-jobs to be executed at other UNICORE sites to the corresponding gateway;
- provide local resource information to the gateway;
- take care of the necessary file transfer;

- schedule sub-jobs and tasks according to the specified work-flow;
- provide job status information and job output.

The NJS assigns a UNICORE job directory to each UNICORE job which serves as a UNIX working space for the job. It is a temporary directory only existing during lifetime of the job at the site. All data needed for job execution has to be imported from permanent file space to the job directory. All data which is needed after the job has finished has to be saved to permanent file space. The user specifies the data to be imported and those to be exported when constructing the job. The transfers are done by UNICORE transparently to the user.

The Target System Interface (TSI) implements the interface to the local operating and resource management system.

2.2 Application Specific Interfaces

UNICORE offers a flexible mechanism to easily integrate existing applications: The plugin interface. It consists of two parts: the first is the interface in the graphical user interface and the second is the corresponding definitions for the software resource and its execution on the target system in the Network Job Supervisor's Incarnation Data Base. These interfaces have been initially exploited for the Car Parrinello Molecular Dynamics application (see Figure 2, and [7] for details).

The plugin mechanism allows for adding new job task elements into the job preparation part as well as components for the output interpretation (i.e. visualization) into the job monitoring part. The core classes and their methods from the basic user client can be exploited by the application specific interface to access user data, model internal dependencies, build the corresponding AJO, retrieve output and alike. Hereby a powerful tool is provided to integrate any further application into the UNICORE framework.

3. Vampir

Performance optimization remains one of the key issues in parallel computing. With the emergence of Grid applications running on more than one system, the task of analyzing and tuning scientific applications actually becomes harder. Tools need to be extended to enable performance analysis also for this new application class in a global Grid environment.

Up to now, the predominant parallel architecture classes have been the classic shared–memory systems with limited scalability and the scalable distributed–memory MPP systems. For each class, performance analysis methodologies and tools (AIMS [14], PABLO [13], PARADYN [9], Paragraph [6], Paraver [8], Vampir [1, 10, 15]) have been developed and many significant scientific and engineering codes have been ported and optimized. Even for clustered SMP

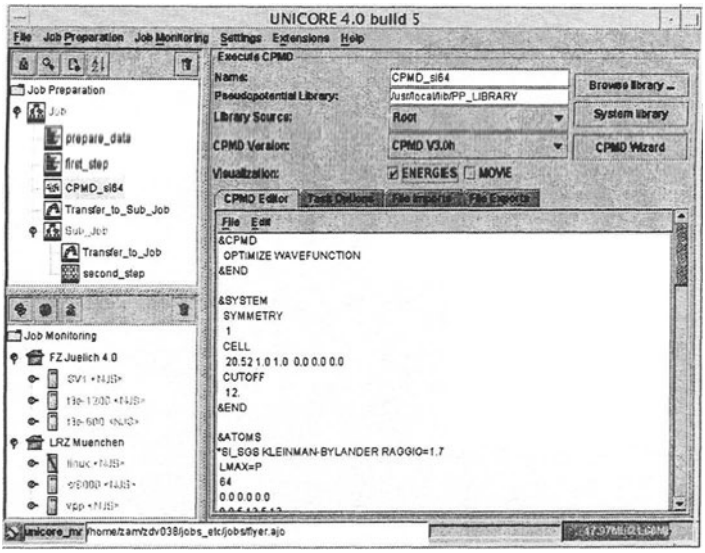


Figure 2. Plugin Interface for CPMD

platforms, this picture has changed. To achieve a good price / performance ratio, these systems do not provide an efficient virtual–shared–memory system, but expose their structure to the applications. An application programmer aiming at scalability beyond one SMP node has to make a choice between the message–passing programming model, or a combination of message–passing between SMP nodes, most probably MPI, and a shared–memory model within one node, most probably OpenMP. In both cases, performance analysis faces new challenges, and there has been already a strong need to extend the current MPP performance–analysis tools by

- support for multi–threading,
- analysis of memory and CPU statistics,
- display of scheduler events and interactions.

The main challenge is an appropriate organization of performance data, both internally to the tools and more important to the end–user: the potentially enormous amount of performance information (in particular if event tracing is used) has to be processed and displayed in a way that an ordinary user can understand.

Vampir [10] is a performance analysis tool which has addressed these kinds of topics for many years. It supports the performance analysis process and makes it easy for the programmer to get insight into the parallel execution

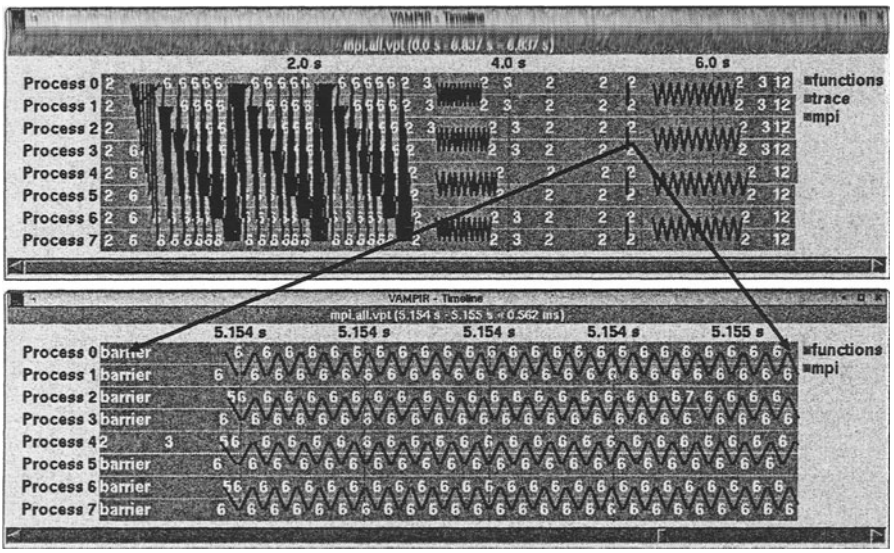


Figure 3. Vampir Timeline with Zooming Capabilities

of a program on any homogeneous parallel system. Vampir converts trace information into a variety of graphical views, e.g. timeline displays (Figure 3) showing state changes and communication, profiling statistics displaying the execution times of routines, communication statistics indicating data volumes and transmission rates, and more. The displays can be related to the source code, and Vampir's advanced navigation functions allow to easily zoom into arbitrary time intervals. The profiling and communication statistics help in identifying performance bottlenecks.

The Vampirtrace profiling tool for MPI applications produces tracefiles that can be analyzed with the Vampir performance analysis tool. It records all calls to the MPI library and all transmitted messages, and allows to define and record arbitrary user defined events.

The performance analysis process gets even more difficult if the program is executed in the Grid where accessibility, connectivity, and security aspects make the situation much more complex.

4. Vampir Integration in UNICORE

To achieve a user-friendly interface which helps the user to generate tracefiles during his application's run, an application specific plugin for Vampirtrace has been integrated into the UNICORE environment.

This interface assists the user in choosing the right configuration settings and preparing the necessary operating system environment for the instrumented

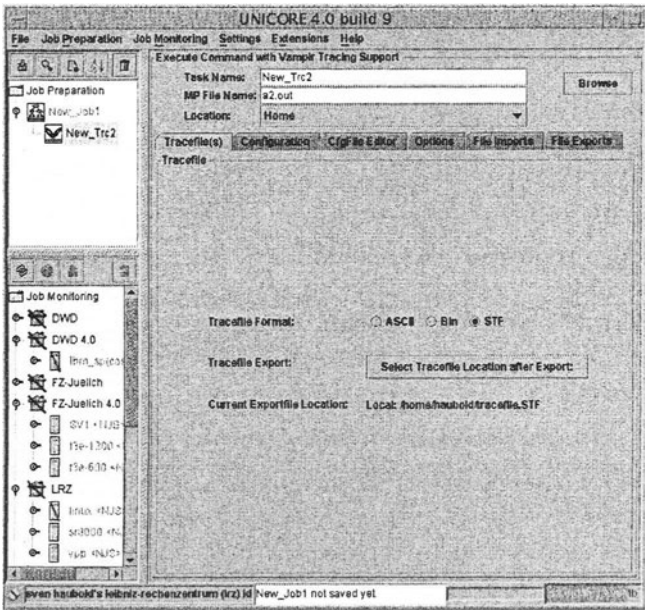


Figure 4. Application Specific Interface for Vampir

program run. Normally, in many environments supporting commercial tools and programs it is not sufficient to have an executable program which was compiled and linked using the appropriate profiling libraries. For example, in order to use Vampirtrace on a system, you must have access to a license key. The license keys are stored in a plain ASCII file. Before the program can be started, the user has to specify where the program can find the necessary license file. This information is given to the application transparently by some environmental variables. Besides, the user has the possibility to specify the extend of information which is written to the tracefiles by some configuration directives placed in a configuration file. With this configuration file, the user can customize various aspects of Vampirtrace's operation and define trace data filters.

The Vampirtrace support plugin has a graphical user interface (Figure 4) which is called from the Job Preparation Menu. In the same way as in the normal Command Task Panel the name and the location of the executable has to be chosen here. Additionally, this panel allows the user to specify the type of the tracefile which has to be generated. The user has to specify the location the produced tracefile has to be moved to after program termination. The Vampirtrace Plugin has a second panel which works as a assistant or wizard (Figure 5).

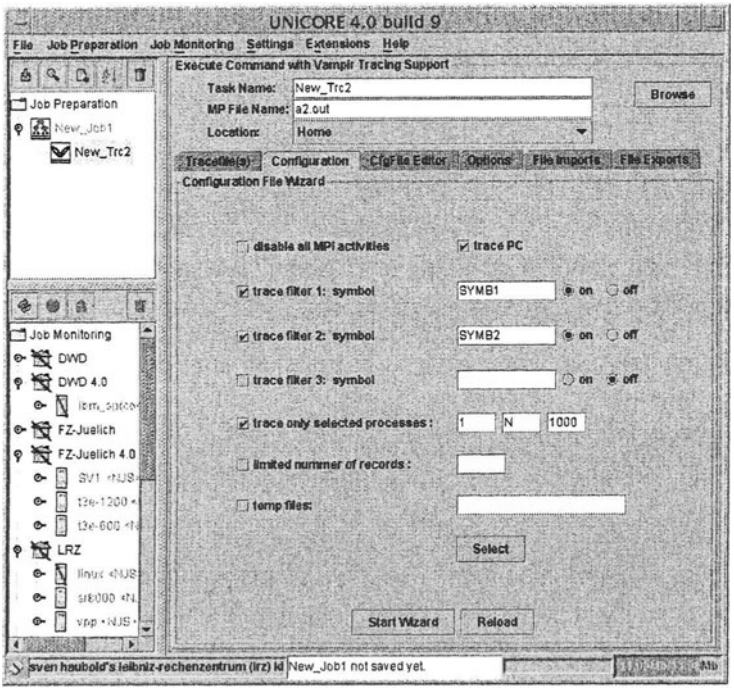


Figure 5. Vampirtrace Wizard

This panel provides some configuration settings often used. These settings can be used also by non-specialists by simple selections. Therefore, it is not necessary that the user knows the exact syntax of the configuration file. Otherwise, for more experienced users it is also possibly to specify all available configuration directives and their parameters in a Configuration File Editor Window (Figure 6).

Besides the specified parameters, the plugin will add the name and type directives for the tracefile automatically to this configuration file. After confirmation of the input, the plugin mechanism will check the correctness of all inputs and produce the necessary configuration file.

As an important second part of the plugin mechanism, all target system specific information like the installed software version or the location of the license file are provided by the system administrator of the Vsite. This information is stored in the NJS Incarnation Data Base and will be added to the AJO during job generation.

So the user can prepare his application to run with simultaneous instrumentation without detailed knowledge of the site or system specifics. The generated tracefile automatically will be merged and transfered to the specified location

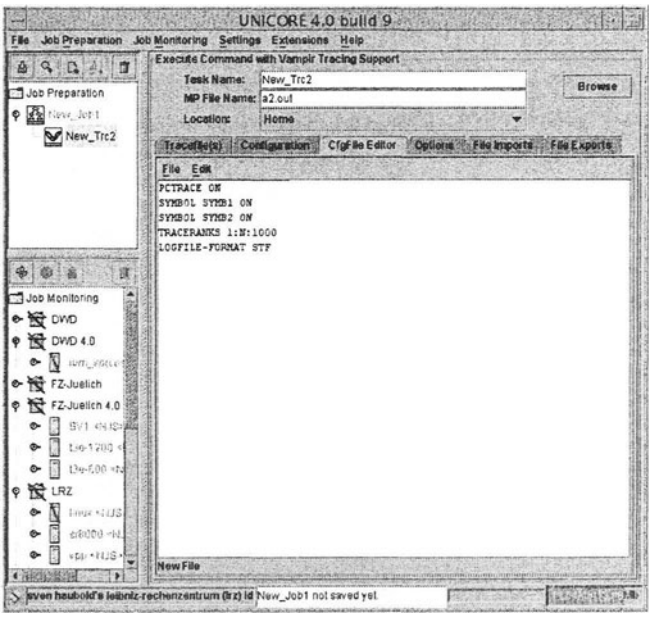


Figure 6. Vampirtrace Configuration File Editor

after task completion and can then be analyzed locally with the Vampir performance analysis tool.

While the graphical user interface of the Vampirtrace support plugin already makes the process of instrumentation easier and more seamless, it becomes still much more helpful if the user intents to analyze Grid applications that are running on distributed or heterogeneous computer systems. Again, using the general plugin mechanism an additional meta-computing plugin (Figure 7) assists the user in submitting and controlling a meta-computing UNICORE job. It consists of two parts. One window determines all settings which are common for all participating target systems. Here, also all the mandatory parameters for the generation of the tracefile can be chosen similar to the previous described Vampirtrace support plugin. The second part gives separately for each target system the names and locations of the executable and the number of processors which should be used on that system. After this the prepared PACX-MPI application can be submitted from the UNICORE client. The UNICORE software itself imports the configuration files to each of the different computer sites before the program is started in the batch systems. After successful program execution, it merges the tracefiles generated during program execution into one common tracefile and exports this to a given destination where it can be analyzed using Vampir.

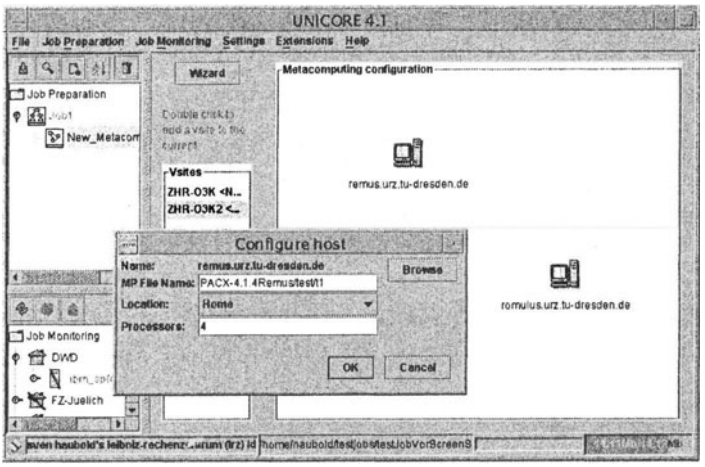


Figure 7. Plugin Interface for Meta-computing Jobs

Figures 8-10 show a test run of an application running on two different SGI Origin 3800 machines (romulus and remus) with 16 processes (8 on each machine).

Figure 8. Vampir Timeline Display Showing an Example Run

Within Vampir, we have added a grouping concept supporting clusters of machines [2]. This is now part of the whole internal infrastructure within Vampir and makes it easier to analyze communication behavior. The fraction of MPI

communications and the different message volume inside the two machines is visible in the *Summary Timeline Display* (Figure 9).

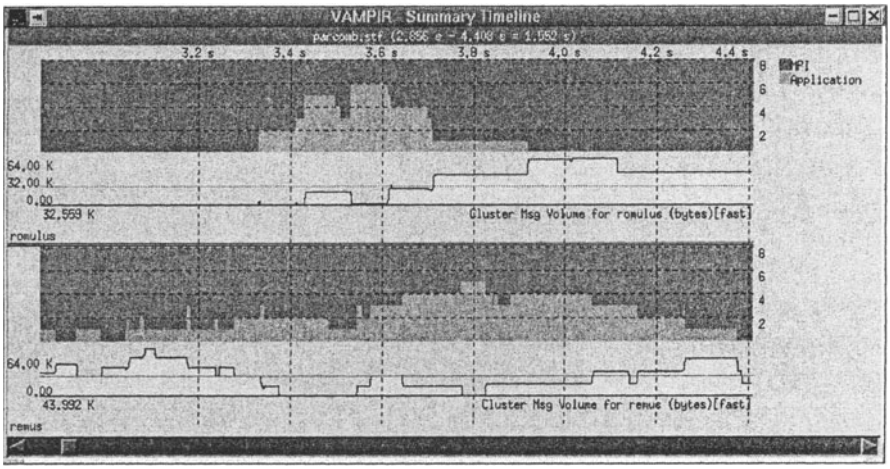


Figure 9. Vampir Summary Timeline

The *Message Statistics Display* (Figure 10) shows fast communication within the machines (values in the diagonal: more than 25 MB/s) and slow communication rates between the systems (less than 1 MB/s). Moreover, the concepts allow to use all well-known features naturally also for Grid applications running distributed between different sites.

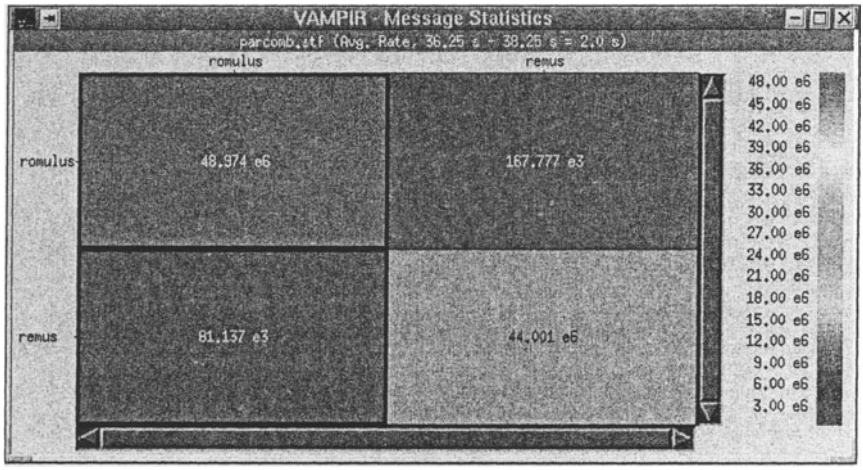


Figure 10. Vampir Message Statistics Display

5. Outlook

Application execution on the Grid is still a challenge, and many scientist are working on a worldwide level to provide functionality and standards. We have shown in which way the UNICORE Grid framework supports performance analysis for complex applications running distributed between different sites. By extending the new grouping concept in the Vampir infrastructure, the full set of Vampir features can now be used also for Grid applications. The focus of the project was the transparent access to the basic trace information via plugins which now allows to ease performance analysis even on a homogeneous system. So far, the UNICORE framework has been proven to support effective performance analysis for distributed Grid applications.

In the near future, the aspects of performance analysis - and the resulting optimization steps - will become one of the major issues addressed by many projects. We will see how general solutions will be to use them also in different environments.

Acknowledgments

Part of the developments described in this paper have been funded by the German Ministry of Education and Research (bmb+f) in the projects UNICORE and UNICORE Plus under Grant 01 IR 703 and 01 IR 001. We also want to thank H.Ch. Hoppe and his team from Pallas GmbH for their support with Vampirtrace and M. Resch and his team from HLRS Stuttgart for the support with PACX and the application PCM.

References

[1] H. Brunst, H.-Ch. Hoppe, W. E. Nagel, and M. Winkler. Performance Optimization for Large Scale Computing: The Scalable VAMPIR Approach. In *Proceedings of ICCS2001*, Springer LNCS 2074, May 2001, pp.0751ff.

[2] H. Brunst, W. E. Nagel, and H.-Ch. Hoppe. Group Based Performance Analysis for Multithreaded SMP Cluster Applications. In *Proceedings of Euro-Par2001*, Springer LNCS 2150, August 2001, pp.148ff.

[3] D.W. Erwin and D.F. Snelling. UNICORE: A Grid Computing Environment. In *Proceedings of Euro-Par 2001*, Springer LNCS 2150, August 2001, pp.825-834

[4] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufman Publishers, 1999

[5] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/research/papers/ogsa.pdf, June 2002

[6] M. T. Heath, A. D. Malony, and D. T. Rover. Visualization for parallel performance evaluation and optimization. In I. J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors, *Software Visualization*, pages 347–365. MIT Press, Cambridge, 1998.

[7] V. Huber. Supporting Car-Parrinello Molecular Dynamics Application with UNICORE. in *Proceedings of the Computational Science - ICCS 2001 International Conference*, San Francisco, May 2001, Part I, pp.560-566

[8] J. Labarta, S. Girona, V. Pillet, T. Cortés, and L. Gregoris. DiP: A Parallel Program Development Environment. In *2nd International EuroPar Conference (EuroPar 96)*, Lyon, France, August 1996.
http://www.cepba.upc.es/paraver.

[9] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, November 1995.
http://www.cs.wisc.edu/~paradyn.

[10] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer 63*, XII(1):69–80, January 1996.
http://www.pallas.de/pages/vampir.htm.

[11] M. Romberg, The UNICORE Architecture: Seamless Access to Distributed Resources. in *Proceedings of the eighth IEEE International Symposium on High Performance Distributed Computing*, pp.287-293, August 1999.

[12] M. Romberg. The UNICORE Grid Infrastructure. *Scientific Programming (Special Issue on Grid Computing)*, 10(2), IOS Press, 2002.

[13] L. DeRose and D. A. Reed. SvPablo: A Multi-Language Architecture-Independent Performance Analysis System. In *Proceedings of the International Conference on Parallel Processing (ICPP'99)*, Fukushima, Japan, September 1999.

[14] J. C. Yan. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 625–633, Wailea, Hawaii, January 1994.
http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS.

[15] Pointers to tools, modules, APIs and documents related to parallel performance analysis.
http://www.fz-juelich.de/apart/wp3/modmain.html.

[16] The crossgrid project http://www.crossgrid.org

[17] Global Grid Forum http://www.globalgridforum.org

[18] The UNICORE project. http://www.fz-juelich.de/unicore

[19] The UNICORE Plus project. http://www.fz-juelich.de/unicoreplus

[20] UNICORE Forum e.V. http://www.unicore.org

[21] The EUROGRID project. http://www.eurogrid.org, funded by the EU, grant period 1.11.2000 - 31.10.2003, grant id IST-1999-20247

[22] The GRIP project. http://www.grid-interoperability.org, funded by the EU, grant period 1.1.2002 - 31.12.2003, grant id IST-2001-32257

Index